Java™ magazine

By and for the Java community

MARCH/APRIL 2017

UI Tools

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

COVER ART BY BOB MORRIS

EDITORIAL

Editor in Chief
Andrew Binstock
Managing Editor
Claire Breen
Copy Editors
Karen Perkins, Jim Donahue
Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo
Design Director
Richard Merchán
Senior Designer
Arianna Pucherelli
Designer
Jaime Ferrand
Senior Publication Designer
Sheila Brennan
Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa
Account Manager
Mark Makinney
Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)
Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

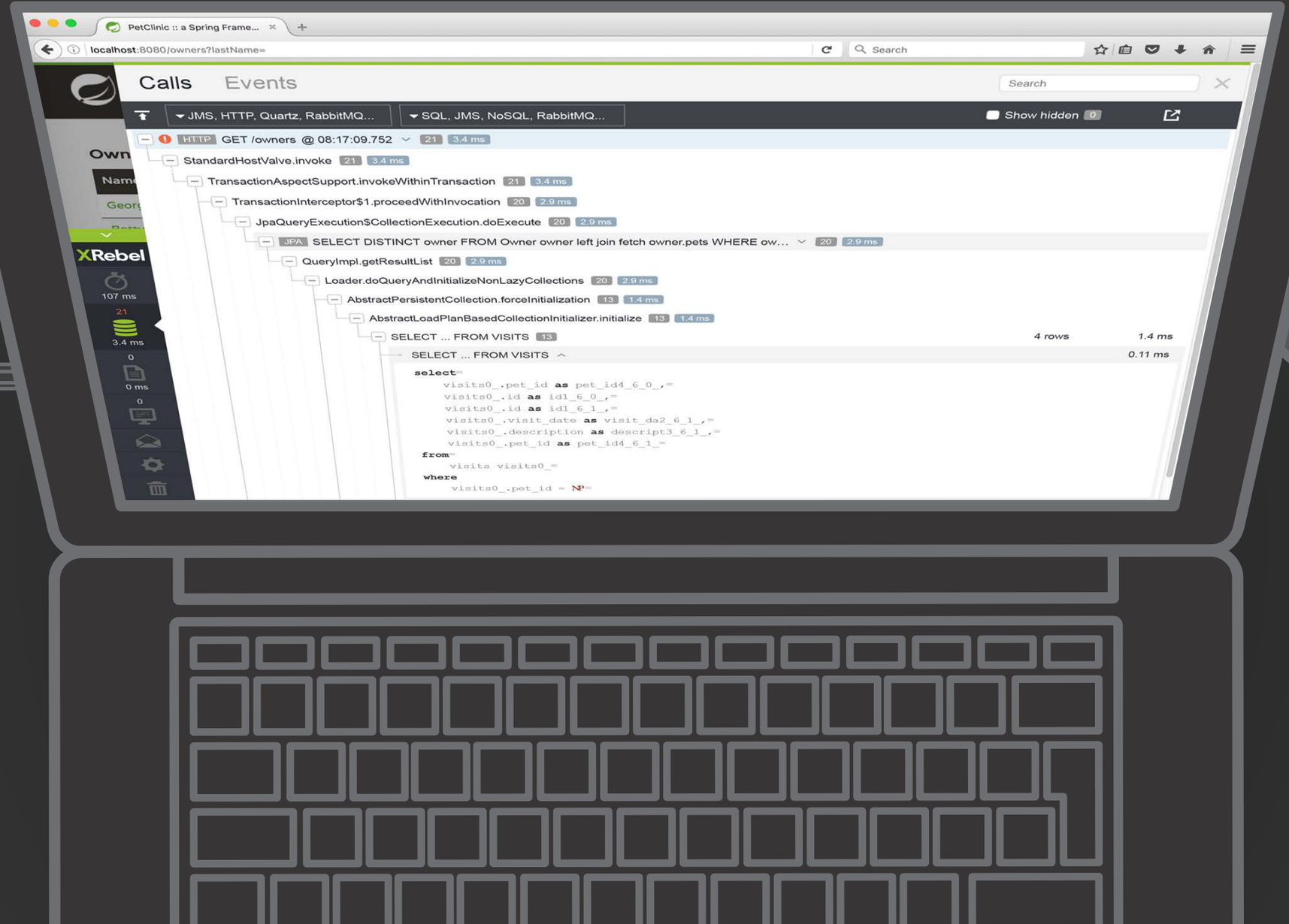
Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

Performance-Focused Developer Tool lets you see request breakdowns, IO queries, logs and exceptions without leaving the window or your webapp.

14 DAY FREE TRIAL!

XRebel

Real time insight, faster apps



ZEROTURNAROUND

The Rise and Fall of JVM Languages

A viable business model is key to language adoption.

For the last 18 months, we at *Java Magazine* have been covering all sorts of interesting JVM languages—from the well known to the obscure. There is no doubt we could continue doing this for another couple of years without covering the same language twice. That's in many ways the glory of the JVM: it is a great platform for language back ends.

The benefits of the JVM include performance, wide availability and familiarity, excellent tools, and thorough documentation. In addition, there's a high level of confidence that the JVM will continue to be widely used, so languages that depend on it won't suddenly need to find a new platform (as those that targeted Adobe Flash, for example, were forced to do).

JVM languages generally fall into two major categories: those that are ports of existing languages (such as the JRuby port of Ruby and the Jython port of Python) and those that are built from the ground up for the JVM (Groovy, Kotlin, Scala, Golo, Fantom, and many others). Those in the latter group often position themselves as an improved alternative to Java the language. And indeed these languages do provide features or syntax that Java has not implemented—often for specific reasons. Other times, the languages lead to Java's adoption of features, in which case the Java team has the benefit of examining those implementations when formulating its own. That Oracle sees value in this dialogue is apparent in its longtime production of the JVM Language Summit at midyear, where JVM language designers come together to compare notes among themselves and with the Java team members.

Because of our long coverage of JVM lan-

PHOTOGRAPH BY BOB ADLER/VERBATIM

guages, I am occasionally asked which of them will become popular enough to "cross the chasm." This term, which originated in Geoffrey Moore's book of the same name, refers to an increase in popularity that drives a technology from the exclusive domain of visionaries and early adopters into the wider embrace of pragmatists and especially of businesses. I believe there are only three languages that are capable of this crossing or have already done so: Groovy, Scala, and Kotlin.

Groovy found success as a quirky scripting language that has filled numerous niches where quick but expressive coding is needed. It is the scripting language for many testing frameworks and is used for writing build scripts in Gradle. It is also unique among the primary JVM languages (the three mentioned above plus Java) in that it did not require corporate sponsorship to become popular. (Even though Pivotal did support it for a few years, Groovy was popular long before Pivotal's acquisition and has continued to be since Pivotal stopped sponsorship.) This is testament to the community skills of the project's longtime leader, Guillaume Laforge.

Today, no language can hope to cross the chasm as Groovy did—that is, without serious financial backing. Writing a language is a very expensive proposition, as is promoting it. While originally an academic creation, Scala was backed by the startup Typesafe until the company realized—as Pivotal did with Groovy—that there is no revenue to be made in selling a new language. As a result, Typesafe changed its name to Lightbend and refocused on its nonlanguage products. The break from being the "Scala company" was so clean that the press release announcing the name change did not even mention the language in the body of the announcement. As I said, there's just no money in languages.

Kotlin relies on a rather different model. The language was devised in part for JetBrains' internal use. Its design is pragmatic and aimed at helping the company reduce costs in developing its extensive line of developer tools. The benefits of developing and promoting Kotlin outweigh its costs and, crucially, JetBrains derives its income from products other than Kotlin. The costs, however, are significant. According to Andrey Breslav at JetBrains, more than two dozen full-time equivalents are developing and promoting Kotlin.

In the process, Kotlin has morphed into more than just an efficiency tool for JetBrains. Its intensely pragmatic orientation has strongly resonated with a significant and active community, which accelerates its movement across the chasm. Kotlin thereby enables JetBrains to bring new developers into its tool ecosystem. But the growing user base also presents the company with the challenge that successful languages often face: managing the demands of users versus the company's own desires for the language.

Because economics support Kotlin's evolution and JetBrains' longstanding knowledge of developers will help it work with the community, I expect that within the next few years Kotlin will fully cross the chasm and emerge as a—or possibly *the*—primary non-Java JVM language, so proving yet again the robustness of the JVM ecosystem.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

JANUARY/FEBRUARY 2017

JavaScript in Java Magazine

We received more than 30 replies to our request in the January/February issue ("The Polyglot Future") for comments about regular coverage of JavaScript. The replies boiled down to three principal points of view, as articulated in the following three notes.

Lots of It!

Yes, please include a regular column on JavaScript. It can be arbitrarily large!

I have been a Java programmer for 16 years. Two years ago, I landed in a job that required client-side JavaScript coding. After a frustrating six months, I learned to respect JavaScript: it is a language and an ecosystem capable of producing quality software and fantastic tooling.

Despite seeing the elegance of libraries and performance of runtimes like Node, I am wondering how "large"-scale development is possible with a language that does not have interfaces, like Java has.

I feel really curious to learn how Java programmers with a similar mindset find their place in a world without interfaces and without a threading API (another topic that fascinates me).

—Csaba Koncz

Some JavaScript, Please

My first thought after reading your editorial was, "I'd read it, but I don't know how much other JVM developers want to hear about the JavaScript ecosystem." But the more I think about it, the more I think it would be a disservice *not* to cover JavaScript. There's a fine line between focusing on one thing and pretending everything else doesn't exist.

That's especially true when JVM-focused languages are compiling to JavaScript. It has been exactly two years since Scala.js went from a science project to something the Scala community is actively promoting. And Kotlin is gunning to be both a Java replacement and a TypeScript replacement, which has meant making Kotlin's type system able to impersonate JavaScript.

I'm not an Android developer, but my wife is an iOS developer who works closely with Android and web developers to keep their code in sync. Naturally they use JavaScript on all three platforms, and I've suggested they look into React Native to help unify the codebase. React is something you could cover without even mentioning JavaScript.

If anything, the argument against covering JavaScript is that there's just too much to do it justice. It seems that every time you dip your toe in, some new framework has taken over. Blink and you miss it.

I'd be fascinated to see a deep dive comparing the tradeoffs between JITs from HotSpot, Dalvik, and the various JavaScript JITs. But maybe that's just me.

—David Leppik

No JavaScript at All

I suggest "Absolutely None" for the introduction of other programming languages in *Java Magazine*. At the same time, I suggest another magazine that covers the polyglot issues discussed in the editorial, with topics involving basic and advanced programming integrated with some database, such as Oracle and MySQL, as well as other programming languages.

—Marcos André Pisching
Professor de Informática no Campus Lages do IFSC
Brazil

Editor Andrew Binstock responds: "All told, the responses were more than 75 percent in favor of inclusion of JavaScript on a regular basis. Based on this, we'll begin regular coverage of it in the next few issues. In the meantime, a JavaScript article that was scheduled long ago is included in this issue. As we move forward, we'll be careful to keep in mind the recommendation of reader William McKenzie, who upon reassurance that JavaScript would not come to dominate the content, kindly replied: 'That sounds appropriate to me. I love [Java Magazine] *and don't want it to lose itself.'"*

Blockchain Blockage

I liked the article "Blockchain: Using Cryptocurrency with Java," in the January/February 2017 issue, but on page 44 the call of the method

```
Greeter.deploy(web3j, credentials,
    BigInteger.ZERO, new Utf8String("..."))
```

is not complete because you need to set the `gasLimit`. What is the value of the `gasLimit` when the `gasPrice` is set to zero?

—Elton DePaula

Author Conor Svensson responds: "It's best if you refer to the accompanying code in the GitHub repository. You can see the correct values to use at this link. If you have any further questions, you can also join the web3j community on Gitter to get assistance."

Being Fair to Gradle Builds

In his article on build tools ("The Design and Construction of Modern Build Tools"), in the January/February 2017 issue, Cédric Beust wrote, "With Gradle, you need to manipulate multiple build.gradle files that, in turn, refer to multiple settings.gradle files."

The only way you would be referring to multiple settings.gradle files is if you're working with multiple projects, even multiple multimodule projects. However, each project, whether a standalone project or a multimodule project, only ever has a single settings.gradle file.

In addition, the text makes it seem like this "new approach" of Kobalt is the only one that makes it possible to "define multiple projects in one build file." This is also possible in Gradle, and it's quite common.

I certainly think it's likely that Kotlin is going to be the main scripting language for Gradle (Kobalt will likely contribute ideas for that), but it's important to be accurate and fair in comparisons of existing procedures.

—David Michael Karr

Author Cédric Beust responds: "You are correct that most projects usually have one settings.gradle file, but that file is pretty much the only way to have multiple modules. Here is the typical way to do this from Gradle's project itself."

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, please indicate this in your message. Please write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

//events /

GREAT INDIAN DEVELOPER SUMMIT

APRIL 25–28
BANGALORE, INDIA
The Great Indian Developer Summit (GIDS), now in its 10th year, offers four days of content grouped by theme. April 26 focuses on Java and JVM languages. Other days focus on web, mobile, DevOps, and big data. Register for each day separately.

PHOTOGRAPH BY AMITH NAG PHOTOGRAPHY/GETTY IMAGES

JavaLand

MARCH 28–30
BRÜHL, GERMANY
This annual conference features more than 100 lectures on subjects such as core Java and JVM languages, enterprise Java and cloud technologies, IoT, front-end and mobile computing, and much more. Scheduled presentations include "Multiplexing and Server Push: HTTP/2 in Java 9," "The Dark and Light Side of JavaFX," "JDK 8 Lambdas: Cool Code that Doesn't Use Streams," "Migrating to Java 9 Modules," and "Java EE 8: Java EE Security API."

O'Reilly Software Architecture Conference

APRIL 2–3, TRAINING
APRIL 3–5, TUTORIALS AND CONFERENCE
NEW YORK, NEW YORK
This event promises four days of in-depth professional training that covers software architecture fundamentals; real-world case studies; and the latest trends in technologies, frameworks, and techniques. Past presentations have included "Introduction to Reactive Applications, Reactive Streams, and Options for the JVM," as well as "Microservice Standardization."

JAX DevOps

APRIL 3 AND 6, WORKSHOPS
APRIL 4 AND 5, CONFERENCE
LONDON, ENGLAND
This event for software experts features in-depth knowledge of the latest technologies and methodologies for lean businesses. The focus is on accelerated delivery cycles, faster changes in functionality, and increased quality in delivery. Conference tracks include agile and company culture, cloud platforms, container technologies, continuous delivery and automation, microservices, and real-world case studies. The conference is preceded and followed by a day of workshops. There's also a two-in-one conference package that provides free access to a parallel conference, JAX Finance.

PHOTOGRAPH BY GUILHEM VELLUT/FLICKR

Devoxx France

APRIL 5, WORKSHOPS
APRIL 6–7, CONFERENCE
PARIS, FRANCE

Devoxx France presents workshops, tutorials, and keynotes from prestigious speakers, followed by a cycle of eight mini conferences every 50 minutes. You can build your own calendar and follow the sessions as you wish. Founded by developers for developers, Devoxx France covers topics ranging from web security to cloud computing. (No English page available.)

IoT Tech Day 2017

APRIL 19
UTRECHT, THE NETHERLANDS

Machine learning and AI, security, wearables, and other smart technologies are among the topics investigated in Europe's biggest IoT-centered conference. One timely track, "Connected Living and Office," examines all aspects of the smart home or office: control of lighting, temperature control, appliances, and security locks for gates and doors. The Things Network—a global community whose mission is to build a global IoT data network—is hosting its popular hackathon at the conference again this year.

Java Day Istanbul 2017

MAY 6
ISTANBUL, TURKEY

With the slogan "By developers, for developers," this conference organized by the Istanbul Java User Group explores Java, web, mobile, big data, cloud, DevOps, and more. It also provides the opportunity for developers to network with tech companies and startups.

JAX 2017

MAY 9–11, CONFERENCE
MAY 8 AND 12, WORKSHOPS
MAINZ, GERMANY

More than 200 internationally renowned speakers give practical and performance-oriented lectures on topics such as Java, Scala, Android, web technologies, agile development models, and DevOps. Workshops are offered on the day preceding and the day following the conference. (No English page available.)

Devoxx UK

MAY 11–12
LONDON, ENGLAND

Devoxx returns to the UK with a focus on Java, web, mobile, JVM languages, architecture, big data, and security. Attracting more than 1,200 attendees, the conference includes more than 120 sessions, with 50-minute conference sessions, three-hour hands-on labs, and many quickie presentations.

Riga Dev Days 2017

MAY 15–17
RIGA, LATVIA

The biggest tech conference in the Baltic States, this three-day event is a joint project of Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Days focuses on the most-relevant topics and technologies for that audience with more than 50 sessions on Java, web, and cloud programming.

GeeCON 2017

MAY 17–19
KRAKOW, POLAND

The GeeCON conference focuses on Java and JVM-based technologies, with special attention to dynamic languages such as Groovy and Ruby. More than 80 conference sessions cover topics such as enterprise architectures, design patterns, distributed computing, software craftsmanship, mobile, and more.

J On The Beach

MAY 17, WORKSHOPS
MAY 18–19, TALKS
MALAGA, SPAIN

JOTB is an international rendezvous for developers interested in big data technologies. JVM and .NET technologies, embedded and IoT development functional programming, and data visualization will all be discussed. Scheduled speakers include longtime Java Champion Martin Thompson and Director of Developer Experience at Red Hat Edson Yanaga.

JEEConf

MAY 26–27
KIEV, UKRAINE

JEEConf is the largest Java conference in Eastern Europe. The annual conference focuses on Java technologies for application development. This year offers five tracks and more than 50 speakers with an emphasis on practical experience and development of real projects. Topics include modern approaches in the development of distributed, highly loaded, scalable enterprise systems with Java, among others.

jPrime

MAY 30–31
SOFIA, BULGARIA

jPrime is a relatively new conference, with two days of talks on Java, JVM languages, mobile and web programming, and best practices. The event is run by the Bulgarian Java User Group and provides opportunities for hacking and networking.

O'Reilly Fluent Conference

JUNE 19–20, TRAINING
JUNE 20–22, TUTORIALS AND CONFERENCE
SAN JOSE, CALIFORNIA

Fluent offers practical training for building sites and apps for the modern web. This event is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers. Training days and tutorials round out the conference experience.

EclipseCon 2017

JUNE 20, "UNCONFERENCE"
JUNE 21–22, CONFERENCE
TOULOUSE, FRANCE

EclipseCon is all about the Eclipse ecosystem. Contributors, adopters, extenders, service providers, consumers, and business and research organizations gather to share their expertise. The two-day conference is preceded by an "Unconference" gathering.

Devoxx Poland

JUNE 21–23
KRAKOW, POLAND

For three days, 100 Java Champions, evangelists, and thought leaders inspire 2,500 developers from 20 different countries at this installment of the popular Devoxx conferences. Tracks on server-side Java, cloud and big data, JVM languages, web and HTML5, and more are on offer. Hacking and networking round out the experience.

QCon New York

JUNE 26–28, CONFERENCE
JUNE 29–30, WORKSHOPS
NEW YORK, NEW YORK

QCon is a practitioner-driven conference for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. The conference covers many different developer topics, frequently including entire Java tracks.

JCrete

JULY 16–21
KOLYMBARI, GREECE

This loosely structured "unconference" involves morning sessions discussing all things Java, combined with afternoons spent socializing, touring, and enjoying the local scene. There is also a JCrete4Kids component for introducing youngsters to programming and Java. Attendees often bring their families.

ÜberConf

JULY 18–21
DENVER, COLORADO

ÜberConf 2017 will be held at the Westin Westminster in downtown Denver. Topics include Java 8, microservice architectures,

Docker, cloud, security, Scala, Groovy, Spring, Android, iOS, NoSQL, and much more.

JavaZone 2017

SEPTEMBER 12, WORKSHOPS
SEPTEMBER 13–14, CONFERENCE
OSLO, NORWAY

JavaZone is a conference for Java developers created by the Norwegian Java User Group, javaBin. The conference has existed since 2001 and now consists of around 200 speakers and 7 parallel tracks over 2 days, plus an additional day of workshops beforehand. You will be joined by approximately 3,000 of your fellow Java developers. Included in the ticket price is a membership in javaBin.

NFJS Boston

SEPTEMBER 29–OCTOBER 1
BOSTON, MASSACHUSETTS

Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 450 events with more than 70,000 attendees. This event in Boston covers the latest trends within the Java and JVM ecosystem, DevOps, and agile development environments.

JavaOne

OCTOBER 1–5
SAN FRANCISCO, CALIFORNIA

Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, Java developers gather from around the world to talk about upcoming releases of Java SE, Java EE, and Java FX; JVM languages; new development tools; insights into recent trends in programming; and tutorials on numerous related Java and JVM topics.

Are you hosting an upcoming Java conference that you would like to see included in this calendar? Please send us a link and a description of your event at least 90 days in advance at javamag_us@oracle.com. Other ways to reach us appear on the last page of this issue.

Oracle Code Events

Oracle Code is a free event for developers to learn about the latest development technologies, practices, and trends, including containers, microservices and API applications, DevOps, databases, open source, development tools and low-code platforms, machine learning, AI, and chatbots. In addition, Oracle Code includes educational sessions for developing software in Java, Node.js, and other programming languages and frameworks using Oracle Database, MySQL, and NoSQL databases.

US AND CANADA

MARCH 27, Washington DC
APRIL 18, Toronto, Ontario, Canada
JUNE 22, Atlanta, Georgia

EUROPE AND MIDDLE EAST

APRIL 20, London, England
APRIL 24, Berlin, Germany
APRIL 28, Prague, Czech Republic
MAY 22, Moscow, Russia
JUNE 6, Brussels, Belgium
JULY 11, Tel Aviv, Israel

ASIA PACIFIC

MAY 10, New Delhi, India
MAY 18, Tokyo, Japan
JULY 14, Beijing, China
JULY 18, Sydney, Australia
AUGUST 4, Bangalore, India
AUGUST 30, Seoul, South Korea

LATIN AMERICA

JUNE 27, São Paulo, Brazil
JUNE 29, Mexico City, Mexico

REGISTER NOW | WWW.DEVOXX.CO.UK

DEVOXX™ United Kingdom

a developer community conference

11th & 12th MAY, 2017

1200 DEVELOPERS

100+ SPEAKERS

110 SESSIONS

11 TRACKS

REFACTORING TO MODERN JAVA: GETTING THE MOST FROM JAVA 8

(LiveLessons video)
By Trisha Gee
Pearson/InformIT LiveLessons

Although this column regularly focuses on books, in this issue we look at a training video sold commercially by the InformIT division of Pearson, the company behind Addison-Wesley and other respected imprints.

This video is a downloadable product (priced around US$100) that consists of DRM-free MP4 files. All told, they represent several hours of high-quality instruction. I watched the videos on my desktop and opened the window to fill the screen so that I could read the code and see the changes easily.

The lecturer is Java Champion Trisha Gee, who is a frequent and well-respected speaker at conferences. Here she presents a variety of refactorings that are available because of the innovations introduced in Java 8. Gee focuses primarily on lambdas, streams, Optionals, and several lesser features. To follow along, you will need to understand these features. The video is squarely aimed at intermediate to advanced developers, and if you don't know the syntax for lambdas or understand what functional interfaces are, you'll quickly find yourself stumbling as you try to follow along. In this sense, the video is tremendously satisfying precisely because it avoids introductory material. That is, you must know Java 8 going in.

After presenting the refactorings, Gee cleverly spends a significant amount of time assessing the performance impact of the code changes. She does this using the Java Microbenchmarking Harness (JMH) and shows that some but not all refactorings improve performance. It's kind of a fascinating process to watch.

Despite the high production values of the video and Gee's practiced delivery (she never stutters or repeats and easily moves backward and forward through subjects to explain a given topic), the video has a few aspects that could bear improvement. At times Gee speaks very fast, which makes it hard to fully understand her point because you're still trying to catch up with what she said 15 seconds ago. My other complaint is that Gee lets the IDE (IntelliJ) find refactorings and implement them, so they're too often done in the blink of an eye. She helpfully shows before and after code in some examples, but it would be easier to follow if we saw her make the changes manually rather than see code instantly transformed by a keystroke. In this regard, although Gee claims that all the refactorings are available in other IDEs, I found that some were not available in NetBeans. I did not verify Eclipse.

Overall, my reservations are about details, not about content, which is uniformly highly instructive and full of immediately applicable information. Gee has posted a less intense and shorter version of this video on YouTube, so you can sample the goods before buying this. I expect you'll be as impressed as I am.—*Andrew Binstock*

Oracle Press™

Your Destination for Java Expertise

Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.

Raspberry Pi with Java: Programming the Internet of Things (IoT)

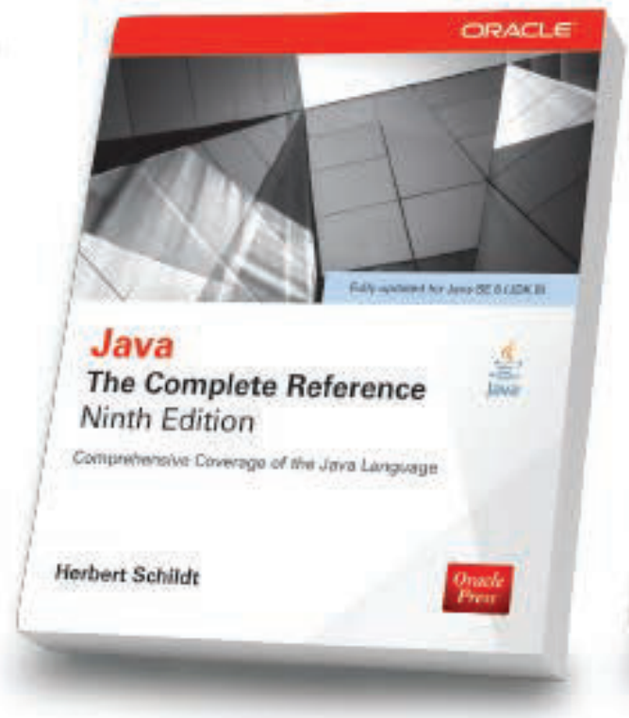
Stephen Chin, James Weaver

Use Raspberry Pi with Java to create innovative devices that power the internet of things.

Introducing JavaFX 8 Programming

Herbert Schildt

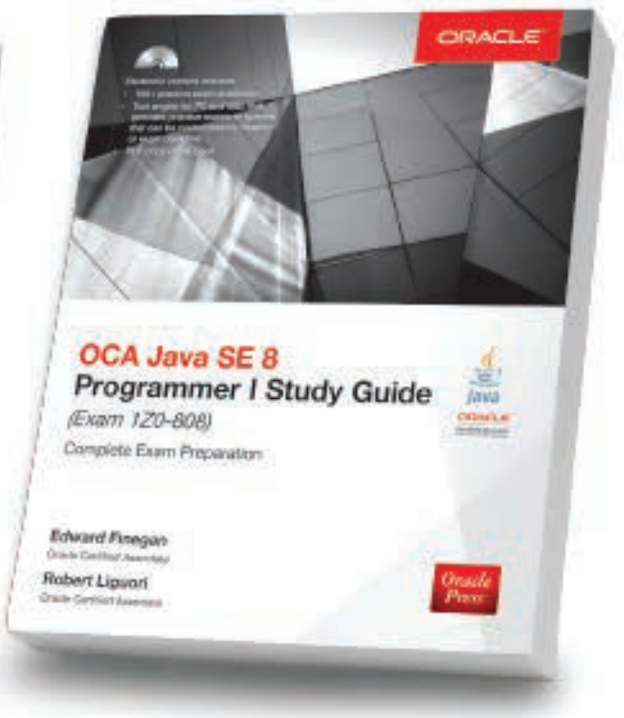
Learn how to develop dynamic JavaFX GUI applications quickly and easily.



Java: The Complete Reference, Ninth Edition

Herbert Schildt

Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.



OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)

Edward Finegan, Robert Liguori

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.

Available in print and as eBooks

Oracle Press™

www.OraclePressBooks.com • @OraclePress

Building User Interfaces

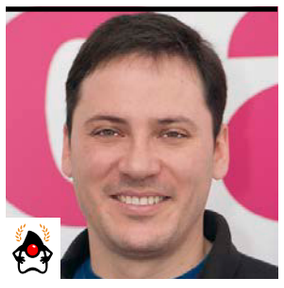
Depending on how your genes are wired, you either love building UIs or find it to be the most annoying task in programming—surpassing even fixing Aunt Thelma's printer because, you know, you're the computer wizard. Regardless of your view of UI development, there can be no denying the now paramount importance of the UI or, put more lavishly, the UX (that is, the user experience). Even the most jaded developers recognize the need for attractive apps with intuitive interfaces. As a result, most of us are now accustomed to designers sitting in on product and app creation and providing feedback and direction as the project unfolds.

Coding UIs used to be an awful chore, with endless minute adjustments having to be constantly recoded. Fortunately, JavaFX greatly facilitated UI construction by scripting it with FXML, which is discussed in our first article (page 17). Our second article (page 24) explores the drag-and-drop design tool Scene Builder, which can generate FXML. Scene Builder was originally an Oracle tool that was released to open source and taken over by Gluon, which has been maintaining it ever since.

Front ends to web applications have their own unique needs, and we cover those too in a pair of articles: one on MVC 1.0 (page 30), a web framework that at one time was considered for inclusion in Java EE 8, and another on a JavaScript toolkit, Oracle JET (page 40), which provides among many resources a large palette of useful controls with easy ways to wire them together.

If UIs are not your favorite topic, we have you covered with a detailed discussion (page 46) of using MQTT, one of the main messaging protocols in IoT. You'll also find an interesting dive (page 60) into how the up-and-coming build tool Gradle uses libraries. And finally, we revisit a topic we've covered before: compact profiles in Java 8 (page 56). In addition, of course, we offer our usual quiz—this time with the inclusion of questions from the entry-level exam—as well as plenty of other goodness. Enjoy!

ART BY BOB MORRIS



ANDRÉS ALMIRAY

Enhanced FXML Using the FXMLLoader

Gain greater flexibility in defining JavaFX UIs declaratively by exploiting the FXML loader mechanism.

The JavaFX UI toolkit delivers a refreshing and modern set of APIs that can be used to build desktop and mobile applications that target the JVM. Similar to its predecessor, Swing, JavaFX provides a standard widget set, as well as the means to extend this widget set with custom components and new behavior.

JavaFX also adds new capabilities such as property bindings, styling support via CSS, and a UI description format named FXML. As the name implies, it's an XML-based format that enables developers to define user interfaces in a declarative way, as opposed to defining interfaces by procedural means—that is, by directly using the JavaFX APIs.

The following benefits are some of the advantages of choosing FXML over the programmatic API:

- FXML is a hierarchical format. The JavaFX SceneGraph is also a hierarchical structure given that it represents the UI elements as a tree data structure. Every node in the SceneGraph relates to a graphical element, such as a button, label, or text field. As a result, it is easier to visualize the component hierarchy in FXML than in plain code.
- FXML can be created on the fly if need be, allowing dynamic UI elements to be added/created at specific points during the application's runtime.
- In many cases, writing FXML results in shorter UI definitions.

You have probably seen FXML before, but in case you haven't, here's a quick sample. The following code snippet defines a grid in which six UI elements are placed in a two-column layout:

```
sample/app.fxml
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.GridPane?>
<GridPane>
  <Label text="Username:"
    GridPane.columnIndex="0" GridPane.rowIndex="0"/>
  <TextField
    GridPane.columnIndex="1" GridPane.rowIndex="0"/>
  <Label text="Password:"
    GridPane.columnIndex="0" GridPane.rowIndex="1"/>
  <PasswordField
    GridPane.columnIndex="1" GridPane.rowIndex="1"/>
  <Button text="Cancel"
    GridPane.columnIndex="0" GridPane.rowIndex="2"/>
  <Button text="Login"
```

```
    GridPane.columnIndex="1" GridPane.rowIndex="2"/>
</GridPane>
```

Figure 1 shows how the application looks when running.

Note that, as in Java code, you must specify `import` statements for the types that are used in the FXML file. These `import` statements serve as a hint to the FXML loading mechanism that's in charge of interpreting the declarative UI description and turning it into a proper SceneGraph with the UI elements in it. This loading mechanism is known as `FXMLLoader`. Using `FXMLLoader` is straightforward, as shown by the following code:

```
sample/App.java
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.net.URL;

public class App extends Application{
    @Override
    public void start(Stage stage) throws Exception {
        URL fxml = getClass().getClassLoader()
                    .getResource("sample/app.fxml");
        FXMLLoader fxmlLoader = new FXMLLoader(fxml);

        stage.setScene(new Scene(fxmlLoader.load()));
        stage.sizeToScene();
        stage.show();
    }
}
```

For now, you only need to tell `FXMLLoader` the location of the FXML resource that you want to load. It's important to remember that the action of creating UI elements and attaching them to the `SceneGraph` must happen inside the UI thread (known as the *FX application thread*). Bad things can happen when you do not follow this rule. Fortunately, the base type `Application` provides a basic lifecycle that ensures code can be called inside such a thread. In this particular case, the `start` method is guaranteed to be called inside the UI thread, which means everything is OK.

Node Properties

If you look closely at the FXML snippet in the first example, you'll see that the FXML nodes `Label` and `Button` define a text attribute. This attribute is in turn mapped to a JavaBean property found in the matching type. Thus when `FXMLLoader` instantiates the first `Label` it encounters, it sets the label's text property to the exact value of the text attribute. In other words, it's as if `FXMLLoader` invoked the following code on your behalf:

```
Label label = new Label();
label.setText("Username:");
// insert label into SceneGraph
```

That's quite a short snippet, and this functionality doesn't seem to be much of an advantage right now; however, take into consideration that UI elements may have several prop-

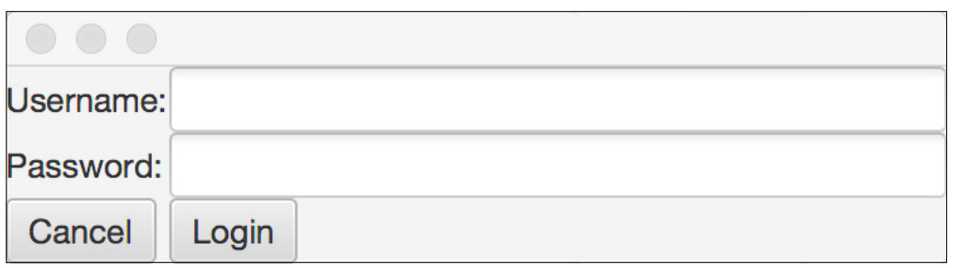


Figure 1. Login screen using FXML

erties that you might want to set. As you continue to define more nodes and attributes, the benefits of FXML will become apparent. It's worth noting that the node type (`Label`, in this case) must define properties matching the JavaBeans convention, resulting in a pair of methods that can be used to query (the getter) and mutate (the setter) the property. The property becomes invisible to `FXMLLoader` if either of these methods is missing. This should be your first clue to understanding why a particular FXML snippet doesn't work in the way you expect—it could be that the property you're attempting to set is not a properly defined JavaBeans property.

You might also be wondering about the additional properties on all six elements that need to be specified to use `GridPane` as a namespace. These properties do not exist on the target nodes themselves, yet they affect the nodes. You can verify the previous statement by changing the values of `columnIndex` and `rowIndex`; depending on the new values, you might end up with a funny layout.

What's going on then? As it turns out, these properties are defined by the `GridPane` class and not the target nodes. This is why the name of the class appears as a namespace. These two properties happen to be defined as static methods on the `GridPane` class, which means they are not JavaBeans properties per se, but part of a convention that FXML enables. As before, each property must have a getter and a setter pair of methods. Additionally each method must have the static modifier. One last part of the convention is that the methods must take the target node as their first argument. This results in the following method definitions found in `GridPane`:

```
public static void
    setColumnIndex(Node node, int index);
public static int getColumnIndex(Node node);
public static void setRowIndex(Node node, int index);
public static int getRowIndex(Node node);
```

You can use the `Node` type as the first argument because it's the supertype of all JavaFX UI elements. This allows `GridPane` to work with any kind of UI element, both those coming from the standard widget set and those provided by third-party libraries. You'll note that the previous method definitions take an integer as a second argument; however, the FXML file defines the corresponding values as literals. That works because `FXMLLoader` is capable of applying type conversions. It does so for all basic types and for enums. One more concern would be storing the value in relation to the supplied node. Again, the JavaFX API comes to your aid because the base type `Node` has a handy properties map that can be used to store any kind of value, and that's precisely what these methods do.

Armed with this new knowledge, you can create your own synthetic properties. *Synthetic properties* are properties that are not explicitly defined in the target type; rather, they relate to the target type by external means, such as a contextual dictionary. Suppose you had the requirement to limit the number of characters a user may type into either the Username or Password field. You could create a new utility class that defines a pair of methods that take a `Node` and a number, using the provided number as a limit. The utility class could look like this:

```
package sample;

import javafx.scene.control.TextInputControl;

public final class InputControlUtils {
    private static final String LIMIT = " limit";

    public static void setMaxTextLimit(
              TextInputControl control, int maxLength) {
      control.getProperties().put(LIMIT, maxLength);
      control.textProperty()
```

```
    .addListener((ov, oldValue, newValue) -> {
      if (control.getText().length() > maxLength) {
          String s = control.getText()
                    .substring(0, maxLength);
          control.setText(s);
      }
    });
}

public static int getMaxTextLimit(
                      TextInputControl control) {
    Object value =
         control.getProperties().get(LIMIT);
    if (value instanceof Number) {
        return ((Number) value).intValue();
    }
    return -1;
    }
}
```

Note that this code does not take into account any invalid values (such as `maxLength` being equal or less than zero), nor does it provide the means to unregister the listener if the `setMaxTestLimit` method is called more than once with the same target node. This code is for illustration purposes and should not be used "as is" in production without additional tweaks.

Moving on, you only need to update the FXML definitions to look like this:

```
sample/app.fxml
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.GridPane?>
<?import sample.InputControlUtils?>
<GridPane>
  <Label text="Username:"
    GridPane.columnIndex="0" GridPane.rowIndex="0"/>
  <TextField
    GridPane.columnIndex="1" GridPane.rowIndex="0"
    InputControlUtils.maxTextLimit="10"/>
  <Label text="Password:"
    GridPane.columnIndex="0" GridPane.rowIndex="1"/>
  <PasswordField
    GridPane.columnIndex="1" GridPane.rowIndex="1"
    InputControlUtils.maxTextLimit="10"/>
  <Button text="Cancel"
    GridPane.columnIndex="0" GridPane.rowIndex="2"/>
  <Button text="Login"
    GridPane.columnIndex="1" GridPane.rowIndex="2"/>
</GridPane>
```

There are two other features to be mentioned when dealing with properties in FXML. The first is the ability to define the name of a default property, which allows you to omit the property name when defining the hierarchy. The name of the property must be defined in code by applying the `@javafx.beans.DefaultProperty` annotation to the target type. Many of the UI elements in the standard JavaFX widget set use this annotation. For example, the `javafx.scene.layout.Pane` type, which is the base type for many widget containers, is annotated with `@DefaultProperty("children")`. This allows you to write FXML as shown in the previous example instead of this:

```
sample/app.fxml
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
```

```
<?import javafx.scene.layout.GridPane?>
<GridPane>
  <children>
    <Label text="Username:"
      GridPane.columnIndex="0" GridPane.rowIndex="0"/>
    <!-- more elements follow -->
  </children>
</GridPane>
```

FXML support in popular IDEs is so well integrated that the IDEs can point out if there's a default property that can be used instead of having it spelled out in the FXML file. Speaking of tooling, it's worth mentioning that you can write FXML by hand, or you can use a free visual tool named Scene Builder, which is described in the accompanying article on page 24.

I should also mention that you have the ability to supply additional hints to `FXMLLoader` when you are instantiating a type. So far, I've shown only types that follow the JavaBeans conventions by providing a no-argument constructor. This is what `FXMLLoader` expects by default. However, sometimes you might encounter a node that defines a constructor that takes arguments. Without the ability to supply additional hints, `FXMLLoader` would simply fail to instantiate the node. You can help it by supplying a name for the constructor arguments as if they were regular properties. To do this, you use the `@javafx.beans.NamedArg` annotation. You must define a name for the argument, and you may define a default value for it if the attribute is omitted in the FXML file. Here's a concrete example found in the standard JavaFX API—the `Insets` class defines the following constructors:

```
public Insets(@NamedArg("top") double top,
              @NamedArg("right") double right,
              @NamedArg("bottom") double bottom,
              @NamedArg("left") double left) {
    this.top = top;
    this.right = right;
    this.bottom = bottom;
    this.left = left;
}

public Insets(@NamedArg("topRightBottomLeft")
              double topRightBottomLeft) {
    this.top = topRightBottomLeft;
    this.right = topRightBottomLeft;
    this.bottom = topRightBottomLeft;
    this.left = topRightBottomLeft;
}
```

This use of annotations enables you to define instances in either of the following ways:

```
<Insets top="10" right="10" bottom="10" left="10"/>
```

or

```
<Insets topRightBottomLeft="10"/>
```

That's all I can say about properties support in FXML for now. Next, let's examine how to configure the `FXMLLoader` itself, using internationalization as an example.

Configuring FXMLLoader for Internationalization

Many languages are spoken in the world today. As a developer, you must take into account the target audience of the applications you build. Sometimes users speak a different language than you speak, so you must be ready to provide a version that's suitable for their needs.

JavaFX widgets are aware of the locale and other regional settings exposed by the JVM, which results in widgets being rendered in right-to-left or left-to-right order depending

on the situation. But you can't say the same about the string literals you add to the application. It would be great if you could somehow externalize those literals and have the FXML files refer to them. Fortunately, `FXMLLoader` has just what you need to solve this problem. You may supply it a `ResourceBundle` at construction time or set the bundle as a property before loading FXML files. You can create this `ResourceBundle` in any way you see fit. My little example, `sample/App.java`, can be updated as follows:

```
ClassLoader cl = getClass().getClassLoader();
URL fxml = cl.getResource("sample/app.fxml");
String resource = "sample/messages_en.properties"
InputStream is =
    cl.getResourceAsStream(resource);
ResourceBundle bundle =
    new PropertyResourceBundle(is);
FXMLLoader fxmlLoader = new FXMLLoader(fxml, bundle);
```

You also define the following pair of properties files: one using English (`en`) as the default locale and another using German (`de`):

sample/messages_en.properties file:

```
username.label=Username:
password.label=Password:
action.cancel.label=Cancel
action.login.label=Login
```

sample/messages_de.properties file:

```
username.label=Benutzername:
password.label=Passwort:
action.cancel.label=Abbrechen
action.login.label=Anmeldung
```

You also must change the FXML file to reflect the keys of the properties you just defined. The convention is to use `%` as a prefix to distinguish a key value from a literal value. Your IDE might even be able to suggest key completions.

```
<Label text="%username.label"
    GridPane.columnIndex="0" GridPane.rowIndex="0"/>
<TextField
    GridPane.columnIndex="1" GridPane.rowIndex="0"
    InputControlUtils.maxTextLimit="10"/>
<Label text="%password.label"
    GridPane.columnIndex="0" GridPane.rowIndex="1"/>
<PasswordField
    GridPane.columnIndex="1" GridPane.rowIndex="1"
    InputControlUtils.maxTextLimit="10"/>
<Button text="%action.cancel.label"
    GridPane.columnIndex="0" GridPane.rowIndex="2"/>
<Button text="%action.login.label"
    GridPane.columnIndex="1" GridPane.rowIndex="2"/>
```

If you run the application again, you should see the same visuals as in **Figure 1**. Now, change the code to load the German version of the properties file, and you'll notice the application looks like **Figure 2**.

Great! These are the first steps to get localization support in an application. As mentioned earlier, you can create the `ResourceBundle` in many ways that are commonly used in regular Java programming. A further step would be to make

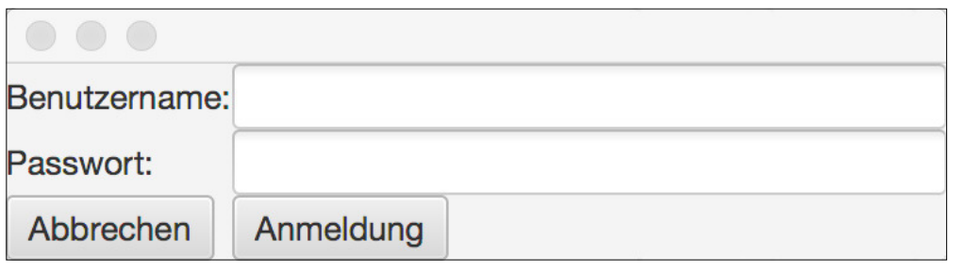


Figure 2. German login screen

the application react to locale changes at runtime. If you choose to do this, you'll need to find a way to let the widgets know that a change of locale has occurred and that the new key values must be fetched again. I'm afraid that `FXMLLoader` alone is not enough, but you have all the basic blocks to get started.

Conclusion

FXML provides a quick and easy way to define UIs declaratively. The tooling support (IDEs and `SceneBuilder`) is good enough to get you started writing FXML with ease. Many of the node types that you can use inside FXML are straightforward, because they follow the JavaBeans conventions. But when they don't, there's still a way to use them with FXML given that the format provides useful extensions and hints to its loading mechanism, `FXMLLoader`. You can also configure localization concerns on widgets via FXML, enabling a wider audience for your applications. In an upcoming article, I'll examine the mechanism exposed by `FXMLLoader` and FXML to further customize and bind the widgets defined in the FXML file—for example, features such as the `fx:controller` attribute and the `@FXML` annotation. `</article>`

Andrés Almiray is a Java and Groovy developer and a Java Champion with more than 17 years of experience in software design and development. He has been involved in web and desktop application development since the early days of Java. He is a true believer in open source and has participated in popular projects such as Groovy, JMatter, Asciidoctor, and others. He's a founding member and current project lead of the Griffon framework, and he is the specification lead for JSR 377.

learn more

Oracle's tutorial on FXML

JOHAN VOS

Scene Builder: The JavaFX UI Design Tool

The interactive UI design tool originally developed by Oracle continues to advance in the open source community.

Scene Builder is a popular tool that enables developers to create Java user interfaces (UIs) using an intuitive drag-and-drop approach. It was initially created and maintained by Oracle and open-sourced. Since March 2015, Gluon has distributed the releases of Scene Builder for Mac, Linux, and Windows systems. In that time, there have been more than 400,000 downloads of the tool. Gluon continues to develop Scene Builder. In this article, I discuss how to use the tool. For readers familiar with Scene Builder from the days when Oracle distributed it, this will serve as both a refresher and an update.

About Scene Builder

UIs for Java client applications can be created programmatically using JavaFX code, or declaratively, using FXML code. The declarative approach requires FXML, which is a format based on XML and well documented by Oracle. While it is possible to create complete UIs writing directly FXML code, it is a tedious process. Fortunately, you can design a UI interactively using Scene Builder, which then translates your UI into FXML that can be used together with other JavaFX code.

In practice, many applications contain both JavaFX code and FXML code. The JavaFX APIs and the FXML constructs are designed to work together.

PHOTOGRAPH BY TON HENDRIKS

How It Works

On the JavaFX code side, the `FXMLLoader` class loads an FXML file from the JAR containing the application file or from the classpath:

```
FXMLLoader.load(getClass().getResource("FXML.fxml"));
```

The connection between your Java code and the UI elements that are declared in the FXML file is made by a controller class. This is a regular Java class that may contain annotations linking the UI elements to Java classes. This approach separates the UI declaration from the behavior of the application, while still allowing the application to access the UI elements directly.

The loader will find the name of the controller class, as specified by `fx:controller="your.package.name.FXMLController"` in the FXML file. Then the loader creates an instance of that class, in which it tries to inject all the objects that have an `fx:id` tag in the FXML file and are marked with the `@FXML` annotation in the controller class. Finally, when the whole FXML file has been loaded, the `FXMLLoader` will call the controller's `initialize` method.

Although the FXML file can be edited within any IDE as a regular XML file, this practice is not recommended, because the IDE provides only basic syntax checking and autocomple-

tion but not visual guidance. A better approach is to open the FXML file with Scene Builder.

Installing Scene Builder

You can install Scene Builder by downloading it. Make sure to download the right version for your operating system. Then follow the platform-specific instructions for installing it in the default location, or select a custom location if you are installing it on Windows. (This is a new feature available in Scene Builder 8.3.0.) Once you have installed Scene Builder, open your IDE so you can set its location and you can open any FXML file from your IDE:

- On NetBeans, select Tools -> Options (or Preferences on Mac) -> Java -> JavaFX, and click Browse to find the main Scene Builder folder.
- On IntelliJ, select File -> Settings (or Preferences on Mac)

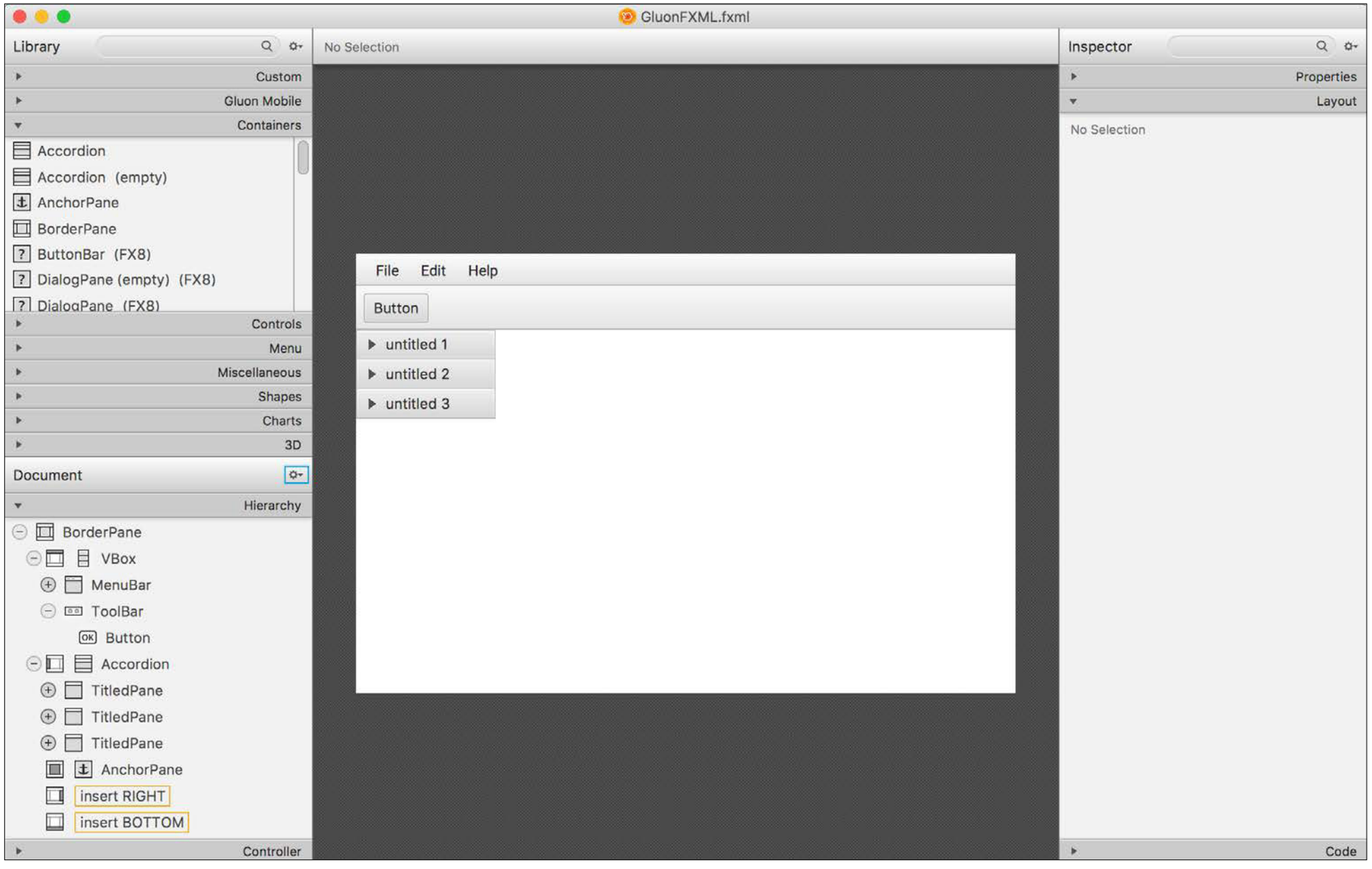


Figure 1. Adding containers and controls

-> Languages & Frameworks -> JavaFX, and browse for the application path.
- On Eclipse, select Window -> Preferences (or Preferences on Mac) -> JavaFX, and browse for the application path.

Whenever you have an FXML file, you will be able to edit it with Scene Builder simply by right-clicking it and selecting Open with Scene Builder.

Creating a Basic Interface

You can open an existing FXML file with Scene Builder, or you can open the Scene Builder application and create a new FXML file.

Creating a UI with Scene Builder is easy. You can drag and drop containers and controls to your view. Let's step through an example.

To begin with, open Scene Builder, and select Start New Project from the Welcome menu. For mobile projects, a built-in Gluon Mobile theme is set. If you want to create a regular JavaFX project, you can do this by selecting the Modena theme from Preview -> JavaFX theme -> Modena (FX8).

Add a main container for your scene. In this example, I will add a `BorderPane`. You can drag a `BorderPane` from the Containers left panel to the middle of the screen or to the hierarchy panel.

In a similar way, you can drag and drop any JavaFX built-in container or control (see **Figure 1**, previous page). You can use the Library Manager to include libraries containing custom controls. The online documentation can help with anything that's not intuitive.

A Hierarchy panel is available. It shows the hierarchy of containers and controls. If you want to use containers and controls in your Java code, you need to tag them with `fx:id`. This can be done in the Code right panel. This `fx:id` tag is a very important concept, because it bridges the world of the designer using Scene Builder with the developer code in an IDE.

If you want to have interactions between the FXML file and your Java code, you need to specify the name of the controller class. This name should be added to the FXML file in the Controller panel. You open the Controller panel by clicking the widget in the lower left corner.

If you want to access UI elements from your Java code (via the controller class), you need to make sure the value provided in the `fx:id` tag is exactly the same as the value of the `@FXML` annotation for the corresponding field in the controller class.

To make this easier for the developer, and to avoid typos, Scene Builder can generate a sample controller skeleton for you. This sample controller is auto-generated Java code that contains FXML-annotated fields for all UI elements that are tagged with `fx:id`.

You can easily copy the different nodes to the controller by selecting View -> Show Sample Controller Skeleton. Click the Copy button, and on your IDE paste the content into the controller class.

Typical code will look like **Listing 1**.

Listing 1. Sample Skeleton

```
import com.gluonhq.charm.glisten.control.AppBar;
import com.gluonhq.charm.glisten.control.Avatar;
import javafx.fxml.FXML;
import javafx.scene.control.ScrollPane;
import javafx.scene.layout.StackPane;

public class GluonFXMLSampe {

    @FXML
    private AppBar appBar;

    @FXML
    private StackPane stackPane;

    @FXML
```

```
    private ScrollPane scrollPane;

    @FXML
    private Avatar avatar;

}
```

You can easily add CSS to the scene by providing a CSS file that can be included using the Stylesheets option in the Properties panel on the right. You can add inline styling also, by providing style rules to any node on the scene. You can apply new or existing style classes to any node as well.

Features related to layout, such as position, dimensions, margin, padding, and transforms (translation, rotation,

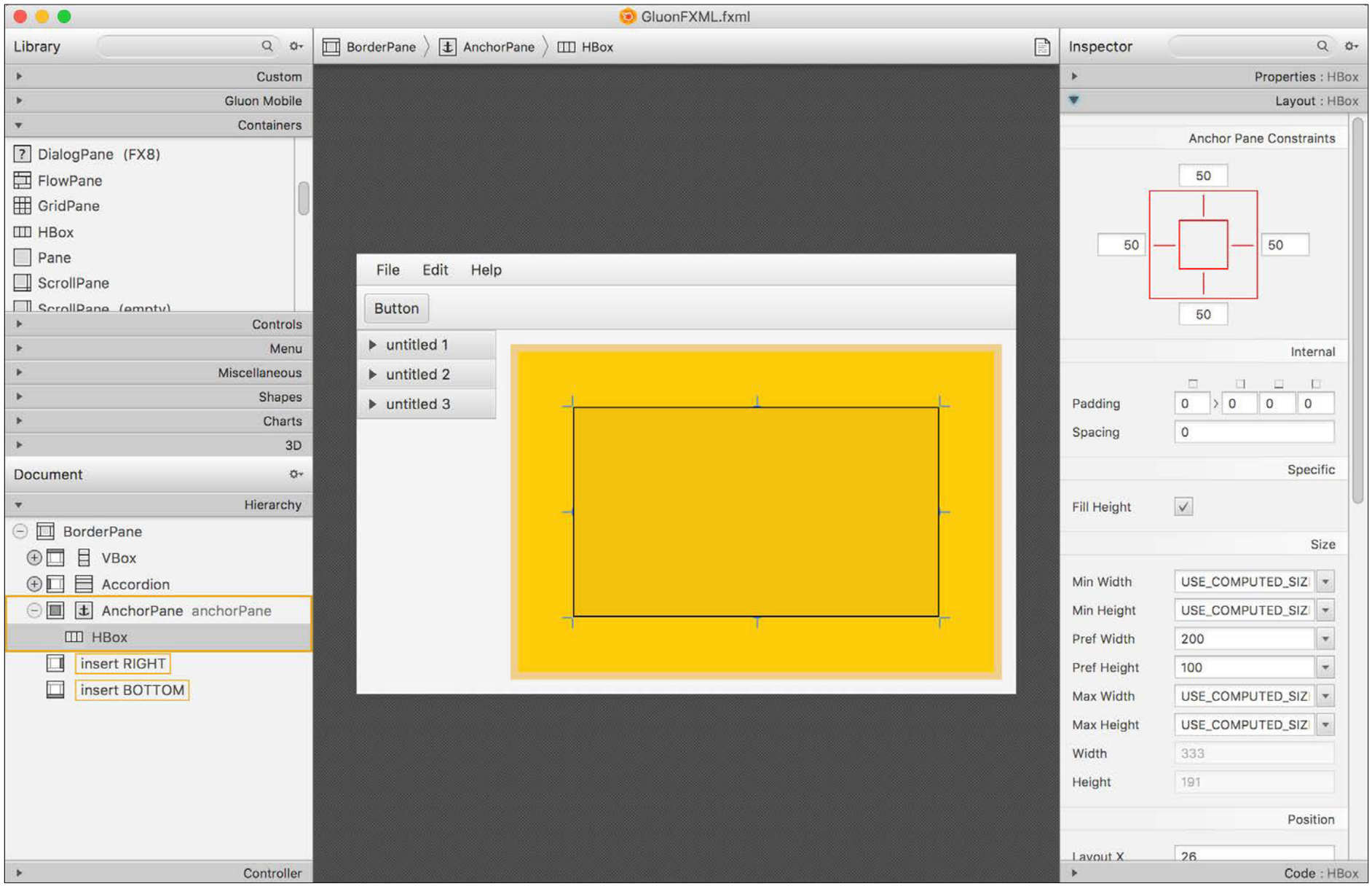


Figure 2. Defining anchor pane constraints in the Layout panel

scaling, and so forth), can be set in the Layout right panel, as shown in Figure 2.

At any moment, you can preview the created scene by clicking Preview -> Show Preview in Window. A resizable dialog box with the designed scene will be shown. By resizing it, you can make sure every node behaves as expected.

Integrate the Basic Interface in a Java App and Show It

Once the FXML is ready, you can integrate it into your Java application by calling `FXMLLoader` to load it. Here's how:

```
public class GluonSceneBuilder extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        Parent root = FXMLLoader.load(getClass()
                      .getResource("GluonFXML.fxml"));

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```

In the controller class, you can add the required action handlers and the response to the user interaction. You can create new controls as well, and combine them with those injected by the `FXMLLoader`. Notice that for the controls you add programmatically, you need to create new instances. This is not needed for controls that are declared in the FXML file, because the `FXMLLoader` already creates them for you. The code snippet below shows a piece of a controller class that works with two controls: an `HBox` control defined in the FXML file, and a `Label` that is not created in the FXML file. The `HBox` instance does not need to be created in the controller class, but the `Label` instance does.

```
@FXML
private HBox hBox;

private Label label;

public void initialize() {
    label = new Label();
    hBox.getChildren().add(label);
    titledPane1.expandedProperty()
      .addListener((obs, ov, nv) -> {
         if (nv) {
            label.setText("TitledPane1");
         }
      });
      . . .
}
```

In a controller class, you can annotate not only fields representing controls with the `@FXML` annotation, but also methods. As an example, I define the following event handler in the controller:

```
@FXML
void buttonClicked(ActionEvent event) {
    label.setText("Button");
}
```

This event handler simply sets the text of the `Label` to `"Button"`. Because the event handler annotated it with `@FXML`, Scene Builder will find it and can assign it to a corresponding action for a node.

Integrate the Interface with Your Favorite IDE

Scene Builder is a standalone application. When FXML is being edited outside Scene Builder (by directly modifying it in your IDE, for example), Scene Builder reacts to the changes

made in the FXML itself or in the CSS files after those changes are saved. Scene Builder also manages to find the controller and the different nodes annotated with `@FXML`.

If you prefer to use Scene Builder fully integrated with your preferred IDE, IntelliJ offers this option by using the version of Scene Builder installed in your system. Note that not all options are supported. For example, there is no support for menus, and custom controls are not allowed.

For Eclipse, there is no built-in integration, but there is a Preview option. On Window -> Show View -> Other select JavaFX -> JavaFX Preview. That option works like the Preview option on Scene Builder, and you can interact with it. In addition, by toggling the Load controller button, it applies the controller and works as if you were running the application embedded in the Preview panel. However, it doesn't work with custom controls.

Community Enhancements in Scene Builder

Although most of the Scene Builder code was contributed by Oracle, the end product is really a synergy between Oracle and the JavaFX community. The code for Scene Builder is now being maintained by Gluon, but it contains contributions from many individuals and companies in the JavaFX community.

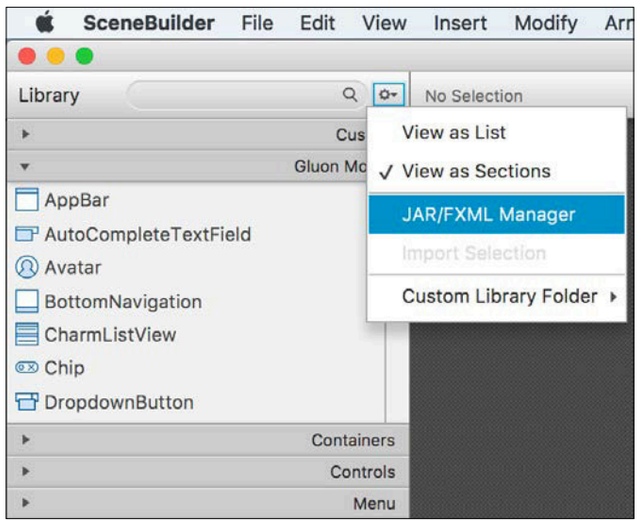


Figure 3. The JAR/FXML Manager menu

As a consequence, many major and minor bug fixes have been applied in Scene Builder, and new functionality has been added. As with any open source project, anyone can report a bug or submit a pull request with a proposed feature or fix.

Among the most recent improvements to Scene Builder are that it now works with an optimized set of imports, and other new features such as a design-time flag. The design-time flag allows developers to provide different behavior for their controls during design time. This ability makes it much easier to work on complex controls, such as controls that at runtime require connections to external resources and databases that are not available at design time. One of the most commonly requested functionalities is better integration with third-party libraries that are available via public repositories. In the past, developers had to manually upload those libraries in Scene Builder. Starting with Gluon Scene Builder 8.2, the Maven Central repository is fully supported, allowing searching, downloading, and installing for any third-party library available in Maven Central or even private repositories. You can access these features by selecting the JAR/FXML Manager option, as shown in **Figure 3**.

Scene Builder then inspects those libraries, and the controls it finds in the libraries are added to the Custom left panel so they can be easily included in the FXML file.

Conclusion

Scene Builder is a tool originally developed by Oracle that enables developers to create UIs using an intuitive drag-and-drop interface. Thanks to the `@FXML` annotation and the ability to assign identifiers and event handlers to controls in Scene Builder, the workflow between Scene Builder and your JavaFX application code is easy. `</article>`

Johan Vos (@johanvos) started working with Java in 1995. He was part of the Blackdown team, porting Java to Linux. In 2015, he cofounded Gluon, which enables enterprises to create mobile Java client applications leveraging their existing back-end infrastructure. Gluon received a Duke's Choice Award in 2015. Vos is a Java Champion and a member of the BeJUG steering group, the Devoxx steering group, and the JCP. He is the lead author of *Pro JavaFX 8* (Apress, 2014).

JOSH JUNEAU

MVC 1.0: A Fresh, New Framework for Enterprise Apps

A look at a remarkably flexible framework that builds on JAX-RS

Applications can be built using several different design architectures. Developers have come a long way from developing web applications using a single JSP page and embedding Java source code directly into the page. Maintenance on such applications was miserable, and they were not much fun to develop. Nowadays, developers have learned that it makes sense to separate code into different portions to promote easier maintenance and development.

One of the most popular application design architectures is model-view-controller (MVC). This architectural pattern separates logic into three distinct categories. The *model* pertains to the database and any code that is used to bind data; the *view* is the code that is used to generate the front end; and the *controller* pertains to the application business logic and intermediate code between the data and the web view. The MVC 1.0 framework follows the MVC design architecture, and it puts the developer into the driver's seat for making low-level design choices and developing custom web controls.

MVC 1.0 was considered for inclusion in Java EE 8 (as JSR 371). Since that time, Oracle has instead sought to move control of the project to the community. In this article, I examine this new framework and explain why I like it, even though there are many other great choices. I also develop an application from the ground up using MVC 1.0 in the NetBeans IDE, allowing you to follow along each step of the way.

Why MVC 1.0?

For many of years, the most popular MVC framework for use with Java EE has been JavaServer Faces (JSF), which follows the MVC architecture and cleanly separates each of the three categories. The JSF framework makes web application development easy, because it takes much of the guesswork out of the design and, thus, allows the developer to focus on the business logic. That said, JSF makes some assumptions for the developer and is not very flexible in some areas. For instance, JSF is very targeted with regard to scoping, and each request follows a complex set of phases through which validation and other tasks occur. JSF can be customized, but there are certain ceremonies that must be followed in order to do so.

One of the key features of JSF is the bundled component architecture, which allows web components to be bundled and made portable for other JSF applications. This abstracts the details of the JavaScript and CSS from the developer, so a tag can simply be placed into the view to render a fully implemented component. JSF has a large ecosystem of component libraries available, which provide many options.

MVC 1.0 does not contain any components, leaving the view completely up to the developer. However, this design does not mean options are not available, because MVC allows just about any view technology to be used to create a front end. The new framework does not contain request phases,

because requests are handled completely by the implementation of controller methods. Given that view options are abundant and the request/response is completely open to custom development, the MVC framework promises to deliver a very flexible web development option. Where JSF is component-based and handles much of the session management for the developer, MVC leaves the developer to create applications using just about any view technology, and it hands over full control of the session management.

Building a Project with MVC 1.0

The MVC framework reference implementation, named Ozark, was defined in JSR 371. This framework is layered on top of JAX-RS. If you are familiar with JAX-RS, you will find it easy to use. If you are not familiar with JAX-RS, you can learn both MVC and JAX-RS reading this article, because they both function very much the same way.

In this article, I walk through the development of the Duke Issue Tracker application, illustrating the basics of the MVC framework along the way. To follow along, you need to have a solid understanding of how Java EE applications are built. I use NetBeans 8.2 here, which already provides support for MVC 1.0, but you can follow along with the IDE of your choice.

Duke Issue Tracker is used to create and view issues for Duke's application. The initial view of the application lists all of the open issues in the upper half of the UI and displays an issue-creation form on the lower half. An Apache Derby database is used to store the data, and this article demonstrates how to use JAX-RS web services and Enterprise JavaBeans (EJBs) to read, create, and update data.

MVC is not quite a production-ready framework. Even though it is fairly developed at this point, it has not yet been completed, nor has it been released as part of any production application server. This means you will need to obtain the Ozark reference implementation JAR file from the project site and deploy it along with your WAR file to a valid container, such as GlassFish 4.1. For the purposes of this example, I deploy to the GlassFish 5 nightly build, and I include Ozark 1.0.0-m02 and MVC API 1.0-edr2 as dependencies in the POM file of the project.

An MVC 1.0 application is simply a JAX-RS application that consists of one or more resources annotated with @Controller.

Server and Project

To get started, create a new NetBeans Maven web application project, and name it DukeIssueTracker. Once the project has been generated, add the required dependencies to the POM, as shown in **Listing 1**. Configure the application to run on JDK 8 and the GlassFish or Payara server of your choice.

Listing 1.

```
<dependency>
    <groupId>javax.mvc</groupId>
    <artifactId>javax.mvc-api</artifactId>
    <version>1.0-edr2</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.glassfish.ozark</groupId>
    <artifactId>ozark</artifactId>
    <version>1.0.0-m02</version>
    <scope>provided</scope>
</dependency>
```

Next, I have created a small set of tables that needs to be installed into the local Apache Derby database or the database of your choice. To do so, run the create.sql file that is packaged with the source code against the database.

The application requires a beans.xml Contexts and Dependency Injection (CDI) configuration file to be placed within the WEB-INF folder. The beans.xml file should configure all beans to be automatically discoverable, as shown in Listing 2.

Listing 2.

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
        "http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
</beans>
```

The views for the application are placed inside the WEB-INF folder because MVC 1.0 searches for view files there and because the contents of WEB-INF are protected by the container. Create a folder named `views` within WEB-INF in which to place the view files. For the purposes of this article, I use Facelets for the view engine, but you can opt for JSP or other view languages if you choose.

An MVC 1.0 application is simply a JAX-RS application that consists of one or more resources annotated with `@Controller`. Because the MVC framework is based upon JAX-RS, application configuration must provide the URL pattern that will be used to access the controller methods or JAX-RS resources. I create a new Java class in a package named `org.dukeissuetrackermvc.config`, and I name the class `ApplicationConfig`. The class should extend the `javax.ws.rs.core.Application` class. I annotate the class with `@ApplicationPath("tracker")` to indicate that "tracker" should be applied to the end of the application path URL to access RESTful or MVC resources. Listing 3 contains the complete code for this class.

Listing 3.

```
package org.mvc.dukeissuetrackermvc;

import java.util.HashMap;
import java.util.Map;
import javax.mvc.security.Csrf;
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
// Optionally this configuration can occur within
// the web.xml deployment descriptor
@ApplicationPath("tracker")
public class ApplicationConfig extends Application {
    // Additional configuration code, if needed
}
```

Entity Classes and RESTful Client

This application uses entity classes that are mapped to the database tables that have been generated for the Apache Derby database. For the purposes of this application, I use the JAX-RS client API to invoke RESTful web services that will provide the data for the application. Use NetBeans wizards to easily generate entity classes from the database and the RESTful application infrastructure.

Once the entity classes and web services have been generated, create a service class that uses a JAX-RS client to read from the web services to obtain data. Create a class in a new package named `org.mvc.dukeissuetracker.service`, and name the class `DukeIssueTrackerService`. This class is a CDI `ApplicationScoped` bean, and it creates a new JAX-RS client in a method annotated with `@PostConstruct` so that the client is generated when the class is constructed. Once the client is created, it will be used to load the data from the RESTful web service into a locally defined list of `DukeIssues` objects. The full listing (due to its length, available from *Java Magazine*'s download area) contains the complete source code for the `DukeIssueTrackerService` class, which also includes

accessor methods that make the JAX-RS client available to the rest of the application.

Controllers

Controllers are responsible for handling web requests and responses for an MVC application. In other words, they bind the view to the data model and provide the necessary business logic for the application. A controller class is a CDI-managed plain old Java object (POJO) that contains both a `@Path` and a `@Controller` (`javax.mvc.annotation.Controller`) annotation at the class level, or has at least one method annotated with `@Path` and `@Controller`. It should also have at least one method annotated as a resource method, using `@GET`, `@PUT`, or `@POST`.

A controller class is instantiated and initialized for each request, so there is no state saved in between requests.

A controller is very much the same as a JAX-RS class, except that it uses different annotations to signify that the controller class or method is used for MVC rather than for JAX-RS. If `@Controller` is placed at the class level, all methods of the class must be used as MVC controllers. However, a controller class can be made hybrid by placing the annotation at the method level. Hybrid classes can contain both MVC and JAX-RS resource methods.

A controller class is instantiated and initialized for each request, so there is no state saved in between requests. To save state or data to display in views, you must use models, which I cover in the next section.

The `@Path` annotation is used to determine which controller class and method is invoked when the URL including the indicated pattern is navigated to from within a browser. The return type of a controller method returns the String name of the view to which the browser will navigate upon return. The default return type of a controller is `text/html`, but it is also possible for the return type to be declared via the `@Produces` annotation.

The code in **Listing 4** demonstrates a simple controller class, which navigates to the `issues.xhtml` view when the URL `http://localhost:8080/DukeIssueTracker/tracker/issues` is used.

Listing 4.

```
@Path("/issues")
@Controller
public class IssuesController {
    @GET
    public String displayIssues() {
        //Perform processing and set Model
        return "issues.xhtml";
    }
}
```

The return type of a controller method determines how the request is processed. Various return types are possible, including Viewable, JAX-RS Response, Redirect, and String. As mentioned before, the default return type is String, but the `@View` annotation can also be used to indicate the view to which the browser should be directed after processing. If it is placed on a method, the return type should be void. Redirects are possible via the use of the JAX-RS Response object's `seeOther()` method, or the client redirect prefix (return `redirect:see/here`).

Building the controller. I now create a class in a new `org.dukeissuetrackermvc.web` package named `IssuesController`. This class is annotated with `@Controller`, and it contains each of the issue tracker resource methods.

Next, I create a method annotated with `@GET` (from `javax.ws.rs.GET`) that will be used for retrieving all active issues in the tracker. I name the method `displayIssues()` and provide a return type of String. The `@GET` annotation is

used to denote a method that returns data to the view. In this case, the database will be queried using the JAX-RS client, which invokes the RESTful endpoint to obtain the issues and stores them in a request-scoped CDI bean named `IssuesBean` (see **Listing 5**).

Listing 5.

```
@Named
@RequestScoped
public class IssuesBean {

    private Issue issue;

    /**
     * @return the issueList
     */
    public List<DukeIssues> getIssueList() {
        return issueList;
    }

    /**
     * @param issueList the issueList to set
     */
    public void setIssueList(
            List<DukeIssues> issueList) {
        this.issueList = issueList;
    }

    private List<DukeIssues> issueList;

    /**
     * @return the issue
     */
    public Issue getIssue() {
        if(issue == null){
            issue = new Issue();
        }
        return issue;
    }

    /**
     * @param issue the issue to set
     */
    public void setIssue(Issue issue) {
        this.issue = issue;
    }

}
```

Listing 6 shows the entire code, this far, for the `IssuesController` class.

Listing 6.

```
@Controller
@Path("/issues")
public class IssuesController {

    @Inject
    private DukeIssueTrackerService
      dukeIssueTrackerService;

    @Inject
    private IssuesBean issuesBean;

    @GET
    public String displayIssues(){
        issuesBean.setIssueList(
         dukeIssueTrackerService.getIssueList());
        return "issues.xhtml";
    }
}
```

The `issues.xhtml` view will be displayed in the browser after the `IssuesBean` is loaded with data.

Models

The *model* of the application refers to the data binding between the data store and the application logic. In the case of MVC 1.0, data can be made available to the application controller, and subsequently to the views, by the injection of a standard `javax.mvc.Models` map or via CDI beans. The preferred technique to make data available is via CDI-based models, which is the technique I use here. However, all MVC 1.0 implementations support the `Models` instance as well.

The `Models` map allows data objects to be made available to the views via a series of key/value pairs in the `Models<String, Object>` map. Much like a standard `HashMap`, the key can be used to access the object, which is the value of the map. In an MVC 1.0 view, the key can be used to obtain access to the data that has been placed into the `Models` object. In the following case, the data would be available via the view by referencing the `dukeIssues` key:

```
…
@Inject
private Models models;
…
public String displayIssues(){
    …
    models.put("dukeIssues",data);
    return "issues.xhtml";
}
```

The default technique, CDI-based models, relies upon the generation and proper scoping of CDI beans to hold data that will be made available to views. `IssuesBean`, shown in **Listing 5**, is used by `DukeIssueTracker` to hold the `List<DukeIssues>` object and make it available to the views. In the case of a CDI bean, the name of the bean or the value provided to the `@Named` annotation is used to reference the data within a view.

Views

The *view* pertains to the presentation layer of the application or the web views. The MVC framework supports two standard view engines out of the box—JSP and Facelets—although others are available. You can also create a new view engine to support any view technology that is not already covered by generating a view engine that implements the `javax.mvc.engine.ViewEngine` interface. A view engine is used to locate and load views, prepare required models, and render the views back to the client. This application uses the Facelets view engine, along with PrimeFaces UI components.

Building the view. I create a folder within the WEB-INF folder and name it `views`. All the views for the application are placed within this folder, because this is the place where the framework searches for view files. Next, I create a new Facelet view named `issues.xhtml` within the newly generated folder.

Facelets is not enabled by default, so to enable it, a web.xml deployment descriptor or a faces-config.xml file must be created for the application. Once that is generated, add the configuration shown in **Listing 7** to enable the Facelets view engine.

Listing 7.

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
      javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
```

```
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

I open the newly created `issues.xhtml` file and add the markup contained in **Listing 8**. (Due to its length, **Listing 8** should be downloaded from *Java Magazine*'s download area or from the projects GitHub repository, which is shown in the links at the end of this article.)

By taking a look at the code, you can see that I can use the CDI bean containing the issues by referencing it via `issuesBean.issueList`, as shown here:

```
<p:dataTable id="issueTable"
    value="#{issuesBean.issueList}"
    var="issue"
    emptyMessage="No issues found">
```

Finishing Up

I've now built the first part of the application successfully. That is, if the application is deployed and the URL `http://localhost:8080/DukeIssueTracker/tracker/issues` is visited, the `displayIssues()` method in `IssuesController` is invoked, and the `issues.xhtml` view is rendered, displaying the list of issues. The form to submit a new issue has not yet been attached to the `IssuesController`, so that needs to be done next.

Looking at the markup in `issues.xhtml`, a form is used to make submissions to `http://localhost:8080/DukeIssueTracker/tracker/issues/create`, so a resource method annotated with `@Path("/create")` must be added to `IssuesController`. I name the method `createIssue` and provide a String return type. The MVC framework uses the `@FormParam` annotation to bind HTML values to Java fields, just like JAX-RS. Therefore, I create a new Java class named `Issue` within the `org.mvc.dukeissuetracker.web` package, and create fields to map against the issue-creation form. **Listing 9** shows an abbreviated form of the completed class. Once it is created, the `Issue` class is passed as a parameter to the `createIssue()` method, and it is annotated with `@BeanParam` to denote that parameter fields are contained within the class.

Listing 9.

```
import java.math.BigDecimal;
import javax.validation.constraints.Size;
import javax.ws.rs.FormParam;
public class Issue {

    @FormParam(value="id")
    private BigDecimal id;

    @FormParam(value="status")
    private String status;

    @FormParam(value="priority")
    private int priority;

    @FormParam(value="requestorFirstName")
    private String requestorFirstName;

    @FormParam(value="requestorLastName")
    private String requestorLastName;

    @FormParam(value="requestorEmail")
    private String requestorEmail;

    @Size(max = 150)
    @FormParam(value="subject")
    private String subject;

    @Size(max = 2000)
    @FormParam(value="description")
    private String description;
```

```
    ...
     //Accessor methods
    ...
}
```

To implement the creation, a `DukeIssues` object should be instantiated within the `createIssue()` method of `IssuesController`, and the fields should be populated with those fields from the `Issue` class. Corresponding `DukeUser`, `DukePriority`, and `DukeStatus` objects should also be instantiated and populated accordingly. Finally, a `DukeIssuesFacade` EJB `createItem()` method is called upon to persist the new record. **Listing 10** shows the source code for the method.

Listing 10.

```
@POST
@Path("/create")
@Controller
public String createItem(@BeanParam Issue form) {
    DukeIssues entity = new DukeIssues();
    entity.setId(new Long(
 dukeIssueTrackerService.getIssueList().size() + 1));

    entity.setDescription(form.getDescription());
    entity.setSubject(form.getSubject());
    DukeUser user = new DukeUser();
    user.setId(Long.valueOf(
            dukeUserFacade.count()+1));
    user.setEmail(form.getRequestorEmail());
    user.setFirstName(form.getRequestorFirstName());
    user.setLastName(form.getRequestorLastName());
    entity.setRequestor(user);
    DukePriority priority = new DukePriority();
    priority.setId(Long.valueOf(
            dukePriorityFacade.count()+1));
    priority.setPriority(form.getPriority());
    entity.setPriority(priority);
    DukeStatus status = new DukeStatus();
    status.setId(Long.valueOf(
            dukeStatusFacade.count()+1));
    status.setStatus("OPEN");
    entity.setStatus(status);
    // Create record
    dukeIssuesFacade.create(entity);
    // Initialize issue list
    dukeIssueTrackerService.setIssueList(null);
    issuesBean.setIssueList(
            dukeIssueTrackerService.getIssueList());

    return "issues.xhtml";

}
```

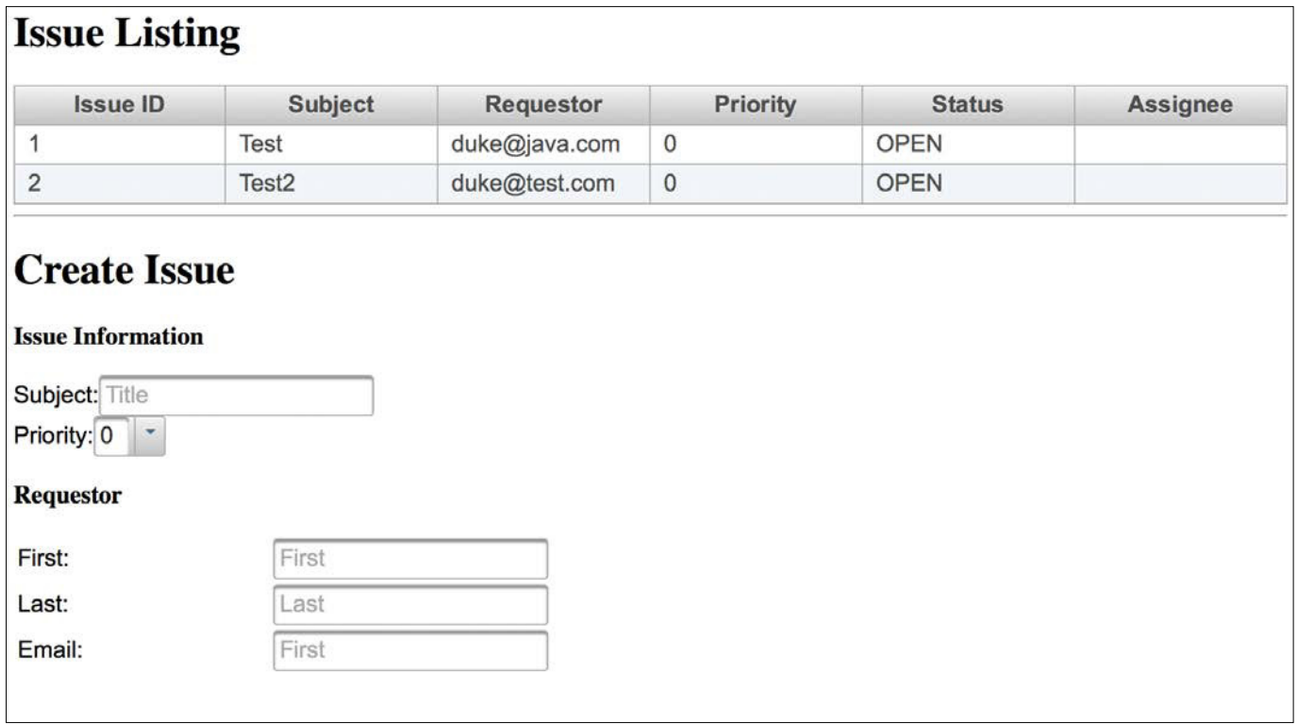


Figure 1. Duke Issue Tracker main view

The final product should look like **Figure 1**.

Validation

Validation can be performed in several ways. Bean validation is performed within the Duke Issue Tracker, because the fields of the `Issue` class are annotated with suitable bean validation annotations. During the creation of an issue, if a field does not pass validation requirements specified in the bean validation constraints, the creation will fail. Annotating a resource method argument as `@Valid` triggers the validation of the bean on every form post.

However, controller methods can be executed even if validation errors are found, as long as they are declared as capable of handling the errors. The `javax.mvc.binding.BindingResult` can be injected into a controller class to indicate that they are capable. The `BindingResult` provides detailed information regarding any validation errors that are encountered. Therefore, custom validation algorithms can be coded into the resource methods if needed.

Exception Handling

Exceptions can be handled on a case-by-case basis using standard try/catch methodology. Often, returning a JAX-RS `Response` object with an error code—as shown below—is the best way to provide the user with information regarding the exception:

```
Return Response.status(
  Response.Status.BAD_REQUEST)
  .entity("error.xhtml").build();
```

It can become cumbersome if the same exception occurs multiple times in an application. In such cases, a global exception handler can be created by generating a class that implements the `javax.ws.rs.ext.ExceptionMapper` class. **Listing 11** demonstrates a global exception handler for `java.lang.Exception`.

Listing 11.

```
import javax.inject.Inject;
import javax.mvc.Models;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class GenericExceptionMapper implements
        ExceptionMapper<Exception> {

  @Inject
  private Models models;

  @Override
  public Response toResponse(Exception exception) {
    models.put("message", exception.getMessage());
    return Response.status(
        Response.Status.BAD_REQUEST)
        .entity("/WEB-INF/error.xhtml")
        .build();
  }
}
```

Listing 11 also demonstrates a clean way of adding messages to the `Models` map to display within the view.

Security

MVC 1.0 controllers are required to support cross-site request forgery (CSRF) validation of tokens. The `Csrf` object can be injected via the `MvcContext` type or via `#{mvc.csrf}` within expression language (EL). A form can be protected by using EL to embed a `Csrf` object into a form, and it will be validated upon submission. The HTTP header can also be used to propagate the `Csrf` object. Automatic validation is enabled by

setting the property `CsrfOptions.IMPLICIT`, or manual validation can be configured by setting the property `CsrfOptions.EXPLICIT` and annotating resource methods with `@CsrfValid`.

Additional Items

There are several ways to pass parameters to resource methods, and this article covered only `@FormParam`. Other techniques are the use of query and path parameters. A query parameter is passed to the method at the end of the URL using the following notation:

```
/controller?param1=value&param2=value
```

Path parameters are passed to the resource method as part of the URL using the following notation:

```
/controller/param1/param2
```

Query parameters can be indicated using the `@QueryParam` annotation, and path parameters can be indicated using the `@PathParam` annotation within the resource method signature. **Listing 12** demonstrates how to use a path parameter in a resource method.

Listing 12.

```
@POST
@Path("/date/{year}/{month}")
public String pathParamDate(
    @PathParam("year") int year,
    @PathParam("month") int month) {
    models.put("specifiedDate",
        month + "/" + year);
    return "showDate.xhtml";
}
```

Observers can be used for logging or other work that must be done when a particular event occurs. An observer can be configured by using the `@Observes` annotation on a resource method parameter, followed by the CDI event to observe.

```
public void onBeforeProcessView
    (@Observes BeforeProcessViewEvent e) {
    // do some work
}
```

Note that CDI observers are fired in a synchronous manner, so long-running tasks should not be handled within an observer.

Conclusion

The MVC 1.0 framework provides a very flexible solution, worthy of serious consideration for enterprise apps. There are numerous excellent posts on MVC 1.0, and I recommend performing web searches to read the bevy of content and examples already available on this new framework. `</article>`

Josh Juneau (@javajuneau) is a Java Champion and an application developer, system analyst, and database administrator. He is a technical writer for Oracle Technology Network and *Java Magazine*, and he has written several books on Java and Java EE for Apress. Juneau is also a JCP expert group member for JSRs 372 and 378.

learn more

- GitHub project source code
- Ozark project site
- MVC specification
- Adding MVC to GlassFish

JOHN BROCK

Building Browser-Based UIs with Oracle JET

A look inside Oracle's open source JavaScript toolkit

Oracle JavaScript Extension Toolkit, or Oracle JET as it's more commonly called, is a modular toolkit based on a collection of open source libraries and open source code contributed by Oracle. It helps JavaScript developers build pure client-side user interfaces that consume and interact with web services such as REST and WebSocket. The toolkit is designed so you can use as little or as much as you need. It's as nonprescriptive in nature as possible.

Why should a Java developer even care about a JavaScript toolkit? With the increasing demand each year for developers to be versed in more aspects of the application stack, it's becoming critical for Java developers to understand the client space as well as the traditional server space. Whether you are responsible for developing REST services that an application will integrate with or the full application, you will inevitably need to know something about how that JavaScript client is written. It could be only for testing purposes, but you will most likely need to know how things work in the browser at some point.

Oracle JET is designed to be a comfortable toolkit for the traditional JavaScript developer. Installation is provided through the Node.js `npm` and `bower` modules, while starter templates are provided by using a Yeoman generator. The steps that follow assume that you are familiar with JavaScript and with Node.js, and that you already have Node.js and Git installed on your system. (Note that Node.js is required only for the installation.)

To install and create your own Oracle JET application, open a terminal or command-prompt window and follow the steps below.

Using `npm`, install the following libraries:

```
npm install -g bower grunt-cli yo
npm install -g generator-oraclejet
```

Once you have the supporting libraries installed, you are ready to generate your first Oracle JET application by typing the following:

```
yo oraclejet <application name> --template=navdrawer
```

This command will create a subdirectory with the `<application name>` that you provided and then install and configure all the required files for the Oracle JET application, using one of many sample application shells as a template. In this case, it will use an application shell that works on both desktop and mobile web browsers. If everything goes correctly, you should see a prompt with wording similar to:

```
Oracle JET: Your app is ready! Change to your new app
```

```
directory [application name] and try grunt build
and serve…
```

Now that you have your application shell created, you can run the sample app and see how it looks in a browser. To do this from the terminal or command-prompt window, type the following commands in your application's root directory:

```
grunt build
grunt serve
```

The browser opens automatically (on localhost at port 8000). The application should show a mostly empty dashboard window when the browser opens.

If you resize the browser window or use the browser tools to render the content in mobile mode, you will see that the application shell is already configured as a responsive application and will change layout depending on the viewport size. Figure 1 shows this same dashboard in a simulated device environment (here, the "device toolbar" in the Developer Tools menu of Google Chrome).

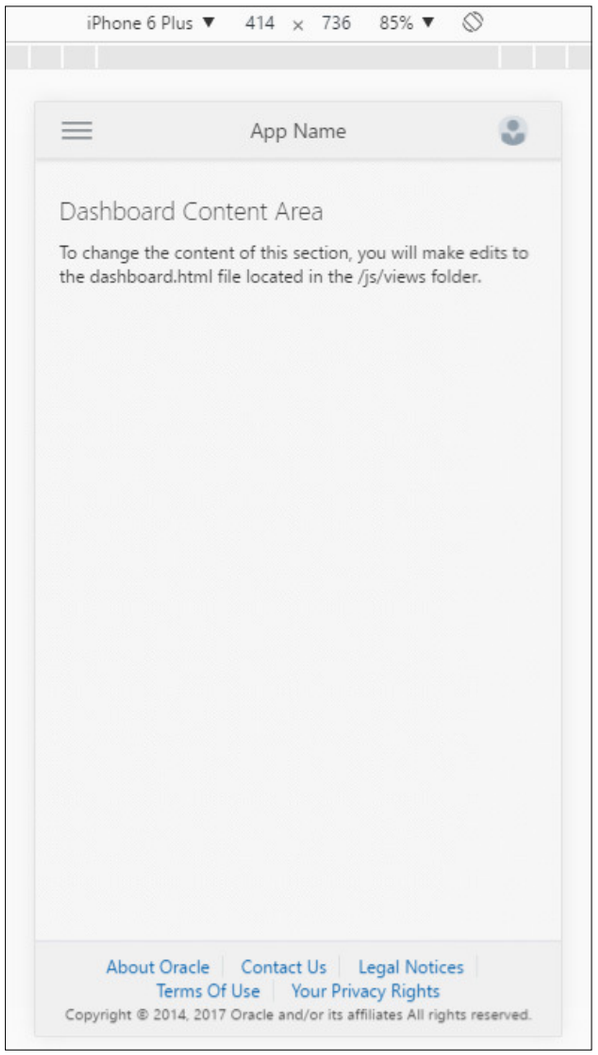


Figure 1. The sample application in a simulated mobile environment

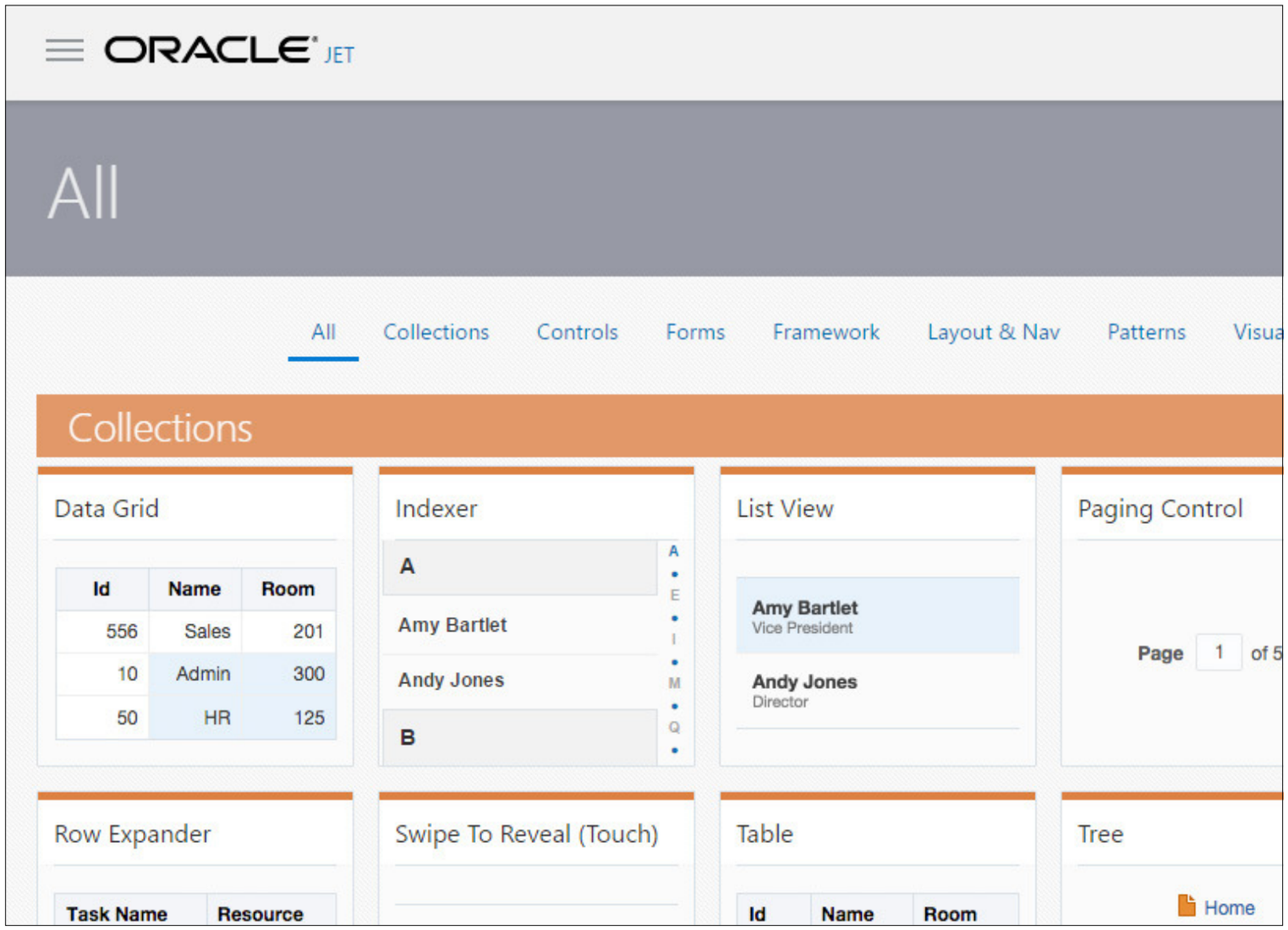


Figure 2. Cookbook samples

Getting started is always the tricky part for anything new. With Oracle JET, you can use a helpful feature of the Oracle JET website called the Cookbook, which has recipes for all kinds of display components and widgets. **Figure 2** shows some samples. The code in the Cookbook is live, so you can make changes to HTML, JavaScript, or CSS and see the changes with a simple click of the Apply button. This is a great way to learn about the API and various options for different UI components.

Once you are ready to add to your own code, you can simply copy and paste the HTML and JavaScript to your own application and continue your development.

For this example, I am going to add one text input field, a selection menu, a button, a radio button set, and a chart. The input field will take a city name; the selection menu will allow you to select a state; and the button will call a REST service to get all of the fueling stations offering alternative fuels in that city and state. The chart will display the returned data. Using radio buttons, you can select whether you want to see the results as a bar chart or a pie chart. You will also be able to click any of the items in the legend of the chart to show or hide that particular item. **Figure 3** is what the final application looks like.

Note that out of the box, all Oracle JET UI components meet the Web Content Accessibility Guidelines (WCAG) 2.0 at the AA level. While this might not mean much to some developers, it should be a goal of all developers to write software that can be used by persons with or without disabilities. Beyond this just being the right thing to do, it's also a requirement for many government and public sector customers, as well as many enterprises around the world.

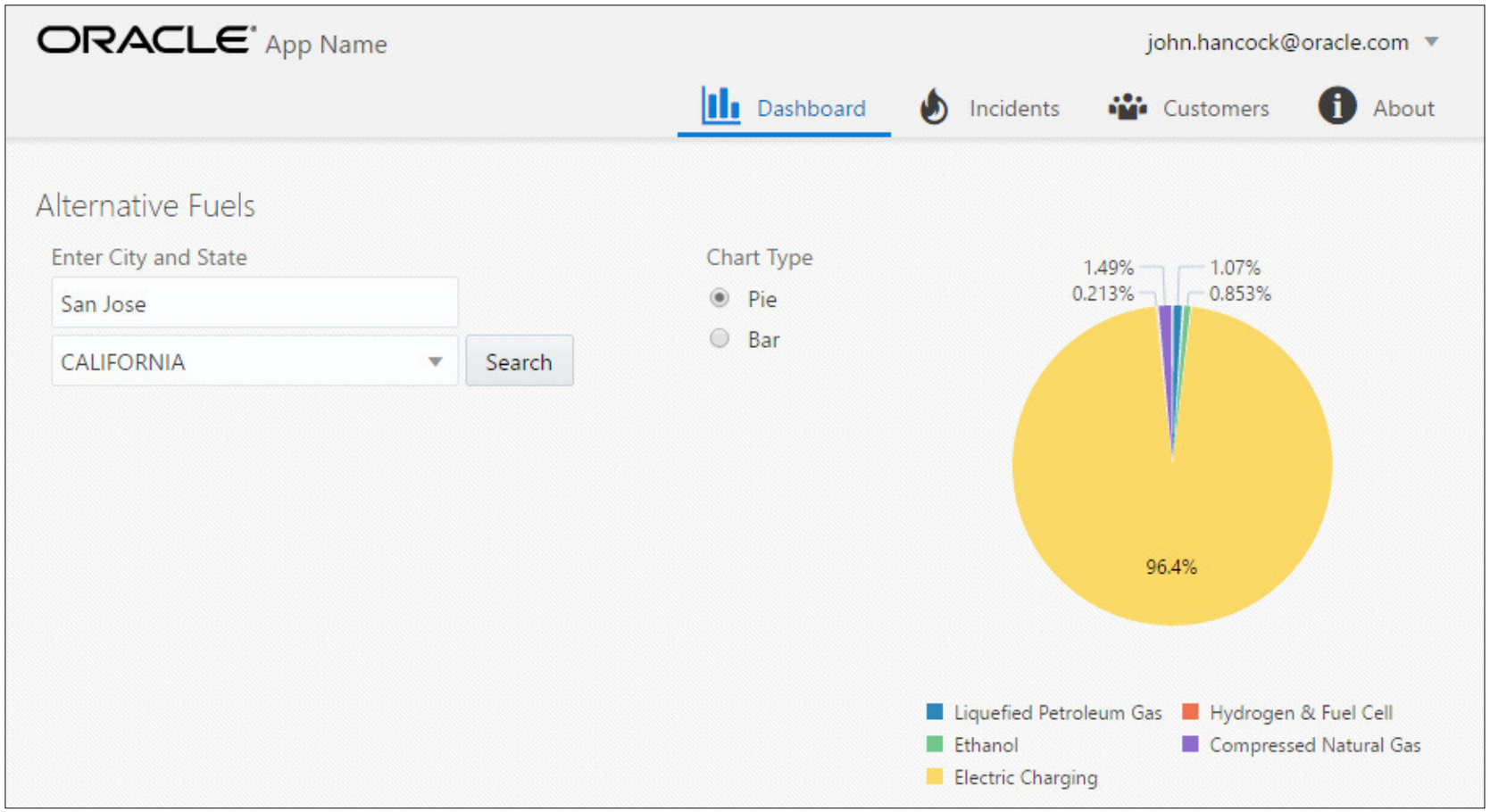


Figure 3. What the sample application will look like

Let's take a look at the code behind the sample application. The architectural structure of an Oracle JET application follows the Model-View-View model (MVVM) pattern. MVVM is derived from the model-view-controller (MVC) pattern and is designed to separate development of the view layer (usually HTML) from the business logic and data layers (usually JavaScript and JSON, respectively). The toolkit is designed to be as

modular as possible, thereby allowing developers to use as little or as much of it as they like. It provides a two-way data binding method based on the open source library Knockout .js. While Knockout is not required by Oracle JET, it is used heavily in samples and documentation as a default binding mechanism. UI components are wrapped as jQuery UI widgets, and a custom Knockout binding integrates those components into the larger MVVM pattern. jQuery UI was chosen because of the hundreds of third-party UI components available for it on the internet. These third-party components can be used with few issues in any Oracle JET–based application.

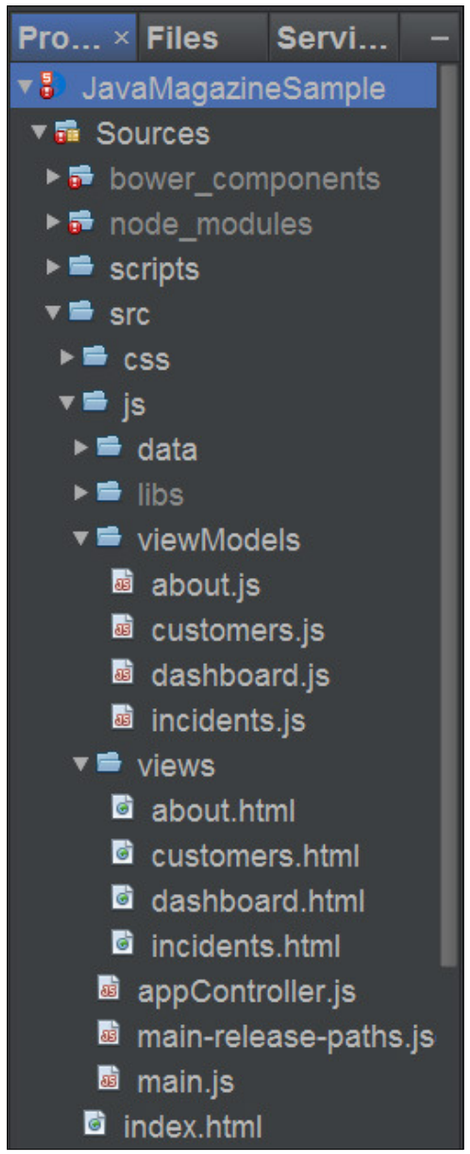


Figure 4. The default directory structure

The inclusion of RequireJS with Oracle JET is to provide lazy loading of resources through the use of the Asynchronous Module Definition (AMD) API. With large enterprise-ready applications, which might have hundreds of JavaScript libraries, it is critical to performance to have the ability to manage the dependencies among those libraries as well as to load only specific libraries when they are needed.

The directory structure shown in **Figure 4** is a common one with JavaScript and CSS in their respective /js and /css folders. Build scripts are kept separate in their own /scripts folder. All the main application code is placed under the /src folder so you can separate what would be included for source code management from what would be platform-specific.

Because Oracle JET is a pure client-side toolkit, the only way to consume and interact with remote data is via web services such as REST or WebSocket.

The starter templates provided by Oracle JET are set up to allow you to build for multiple platforms. You can build a web or mobile hybrid application from the same source code. Currently Android, iOS, and Windows are supported mobile platforms. At build time, native themes are added for the platform that you have selected, so the application's form elements and other platform-specific theming automatically look like those of a native app. You can see a lot of this in action by changing the themes in the Cookbook on the Oracle JET website.

In addition to the accessibility features mentioned earlier (which also carry over to mobile apps), all Oracle JET UI components are touch- and gesture-enabled out of the box. If you have a touchscreen on your laptop, Oracle JET–based applications will work properly.

Looking at the code itself, you will notice that the index .html file provides the shell of the application, while the main content is actually loaded using a feature called ojModule. This design enables developers to think of their pages as separate modules that are loaded into the content area asynchronously when they are needed. This approach allows for more-distributed work on an application, because different teams can work on different modules without stepping on each other's code. An ojModule binding consists of its own HTML (view) and JavaScript (viewModel) file. As shown

in Figure 4, there are separate directories for views and viewModels as part of the app structure. In this sample application, all code was added to the dashboard module (dashboard.html and dashboard.js files).

The dashboard view file is where you focus on writing your HTML code. Oracle JET is what is often referred to as an *HTML forward toolkit*. This means that you use the semantics of HTML5 itself to provide layout and structure to your application. This is in contrast to a *JavaScript forward framework*, where you would write JavaScript to generate your HTML. The business logic is handled in the dashboard viewModel file.

The following code excerpt is an example of what the view looks like:

```
<div class="oj-flex-item">
  <label id="groupLabel">Enter City and State</label>
  <div role="group" aria-labelledby="groupLabel"
    class="oj-form-control-group">
    <input id="text-input" aria-label="city"
      type="text"
      data-bind=
        "ojComponent: {component: 'ojInputText',
                       value: cityVal}"/>
    <input id="text-input" aria-label="state"
      type="text"
      data-bind=
        "ojComponent: {component: 'ojSelect',
                       options: States,
                       value: selectVal}"/>
    <button data-bind=
      "ojComponent: {component: 'ojButton',
                     label:'Search'},
                     click:getData">
    </button>
  </div>
</div>
```

Notice that the `getData` method is bound to the `click` event for the Search button. The values for the input field and selection menu are also bound to values that are defined in the viewModel. This data binding functionality is provided by Knockout.

Looking at the viewModel, you can see that the Knockout observables are defined, along with their default values in the following code:

```
self.handleActivated = function (info) {
  self.cityVal = ko.observable('San Jose');
  self.chartType = ko.observable('pie');
  self.selectVal = ko.observableArray(['CA']);
  self.pieSeriesValue = ko.observableArray([]);
  self.groupsValue = ko.observableArray(['Fuel Types']);
  self.seriesValue = ko.observable();
  . . .
```

Knockout observables are special variables to which Knockout attaches listeners for the purpose of two-way data binding. You will recognize these variables because they are defined by `ko.observable` or `ko.observableArray`. If you change the value of one of these variables, anything bound to it on the view side of the application is automatically updated to the new value. The reverse is also true if you have bound the value of an editable component such as an input field. Once the user updates that field, the variable is updated and anything else bound to that same variable will be notified and updated as well. This is what two-way data binding is all about. You don't need to worry about doing any kind of Document Object Model (DOM) manipulation in your JavaScript code to update the value of your UI components.

The `getData()` method that is called when the Search button is clicked is shown in the following code:

```
self.getData = function () {
```

```
// using a Promise to allow the chart to
// render only once the data is available.
self.seriesValue(new Promise(
 function (resolve, reject) {
   var url =
    "https://api.data.gov/nrel/alt-fuel-stations" +
    "/v1/nearest.json?location=" +
    + self.cityVal() + "+" + self.selectVal()+
    "&api_key=<your api key goes here>";
   $.getJSON(url).then(function (data) {
     var fuels = data.station_counts.fuels;
     var seriesData = [];
   for (var prop in fuels) {
     if (fuels[prop].total > 0) {
       seriesData.push({name: getFuelName(prop),
         items: [fuels[prop].total] })
     }
     resolve(seriesData);
    });
  }))
};
```

This code uses a JavaScript `Promise` to return the results of the call to the REST service after the data is actually ready. The `Promise` object is a new addition in the ECMAScript 2015 language specification (also known as ES6) that helps when you are working with asynchronous data such as REST calls.

Because Oracle JET is a pure client-side toolkit, the only way to consume and interact with remote data is via web services such as REST or WebSocket. You can use any method you like for making the web service call. The most common way is to use jQuery Ajax methods or `jQuery.getJSON`, but you can use plain JavaScript `XMLHttpRequest` API calls as well. For more-complex data interactions—such as those used in applications that need to create, read, update, and delete (CRUD) data—Oracle JET provides a Common Model API that simplifies working with multiple types of data sources. You can read more about the Common Model API in the "Framework" section of the Oracle JET Cookbook.

> Oracle has used Oracle JET as the foundation for **more than 70 percent** of its new cloud service offerings.

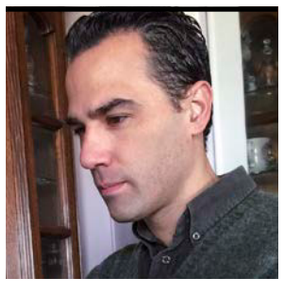
If you would like to take a closer look at the sample application and run it in your own development environment, you can clone the project from GitHub. Installation and configuration steps are included in the README file for the project.

Conclusion

While Oracle JET has actually been around for more than three years, it has been available to the open source community for only about one year. Oracle has used Oracle JET as the foundation for more than 70 percent of its new cloud service offerings—so you know that it meets the needs of very large enterprise-scale applications.

Oracle JET is most powerful when you understand the concepts and ideas behind its creation. To get that understanding, you can read the developer's guide or take the free online training course (login required). Both are great resources for beginners and developers. For announcements and interaction with others using Oracle JET, follow @oraclejet on Twitter or engage with other developers on the Oracle JET Community site. `</article>`

John "JB" Brock has spoken at JavaOne, Oracle OpenWorld, and many Developer Days events over the last decade. He is a coauthor of *Java EE and HTML5 Enterprise Application Development* (McGraw-Hill, 2014), a two-time winner of the JavaOne Rock Star Award, and the senior product manager for Oracle JET.



GASTÓN HILLAR

Simple Messaging with MQTT

Use the principal IoT messaging protocol to asynchronously send and receive data from devices—in this case, from drones.

MQ Telemetry Transport (MQTT) is a lightweight publish-subscribe messaging protocol, especially suitable for small devices but also useful for any device that requires messaging over a network. In this article, I describe how to publish and receive messages with Java through the Mosquitto broker with the asynchronous API provided by the Eclipse Paho project Java client.

In this article, I develop and explain a project in which three drones use a Mosquitto broker to publish and receive text messages. Some of these messages provide only information to a channel and others specify a command and a destination by which they tell specific drones to display their altitude in feet. I use the asynchronous methods included in the Eclipse Paho Java Client to connect to the broker, publish messages, and subscribe to topics. This way, you can see how to work with MQTT in Java using a non-blocking behavior.

The Pieces of the Puzzle

I will shortly explain how to include the necessary references to work with the latest version of the Eclipse Paho Java Client. However, before moving forward, it is necessary to understand the different pieces of this puzzle: the MQTT protocol, Mosquitto, the Eclipse Paho project, and Eclipse Paho Java Client.

The MQTT protocol is a machine-to-machine (M2M) connectivity protocol used extensively in the Internet of Things (IoT), and it is gaining popularity in mobile and web applications. MQTT is a protocol that works with a publish-subscribe mechanism and runs on top of the TCP/IP protocol. It is lighter than the HTTP protocol and, therefore, it is a very interesting option whenever you need to send and receive data in real time with a publish-subscribe model and you need the lowest possible footprint. However, as always happens, the reduced footprint comes at a price: MQTT does not offer great extensibility.

Mosquitto is an open source message broker that implements two versions of the MQTT protocol: 3.1 and 3.1.1. You can use Mosquitto to make any device subscribe to a specific channel, known as a *topic* in MQTT terminology. All subscribed devices will receive all the messages published by other devices to this topic. Mosquitto, with its publish-subscribe model, is an iot.eclipse.org project, also known as Eclipse IoT, and it is provided under the Eclipse Distribution License (EDL). It is important to know that, at the time of this writing, MQTT version 5 has reached the working draft stage.

The Eclipse Paho project offers an open source implementation of an MQTT client library that is capable of working with the same two versions of the MQTT protocol supported by Mosquitto: 3.1 and 3.1.1. The Eclipse Paho Java Client provides both a synchronous and an asynchronous API. As previously explained, I will demonstrate how to work with its asynchronous API. However, bear in mind that there is

a synchronous version you can use if you don't need the nonblocking features and you want to keep your code simpler.

It is important to understand that the MQTT connection is always between a client and the broker—in this case, between Paho Java Client and Mosquitto. You will never connect one client to another client through a direct connection. The dialogue is always between a client and the MQTT broker. In my example, each drone establishes a connection with Mosquitto, and a master drone sends messages with commands that are meant to be processed by specific drones.

You will never connect one client to another client through a direct connection. **The dialogue is always between a client and the MQTT broker.**

It is possible for any client to subscribe to a specific topic. By doing so, it will receive all the messages published to that topic. In addition, the client can publish messages to that specific topic or to other topics. In my example, I work with just one topic, and I won't take advantage of the wildcards that allow you to work with many topics at the same time. However, once you understand how to work with the Java client, you can use the sample code to take advantage of the additional features, based on your specific needs.

I don't want to focus on the configuration of a Mosquitto message browser. Instead, I want to put my efforts into demonstrating how to publish and receive messages. Eclipse allows you to use a publicly accessible sandbox server for the Eclipse IoT projects at iot.eclipse.org port 1883 identified with the following URI: tcp://iot.eclipse.org:1883. I will use this sandbox server as the Mosquitto message broker, without any security. That way, you don't have to lose time setting up a Mosquitto message broker to test the example. Of course, a real-life application would require you to set up a Mosquitto message broker. I provide many useful links to allow you to learn more about Mosquitto and MQTT at the end of this article.

You need to make sure that your software and hardware firewalls allow the application or the IDE to work with TCP in port 1883.

Using the Java Client for MQTT

At the time of this writing, the latest release of Paho Java Client is 1.1.0. I will use this version to include the necessary dependencies. The easiest way to use the Java client is to start a Maven project in your favorite Java IDE and add the following lines before `</project>` in the pom.xml file. I'm assuming that you are working with an empty Maven project. If you have other repositories or dependencies, make sure you edit the pom.xml file to include the new entries, as shown next. You can also use the features included in your favorite IDE or its Maven plugins to add the repository and the dependency.

```
<repositories>
  <repository>
    <id>Eclipse Paho Repo</id>
    <url>https://repo.eclipse.org/content/
         repositories/paho-releases/</url>
    </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.eclipse.paho</groupId>
    <artifactId>
      org.eclipse.paho.client.mqttv3
    </artifactId>
    <version>1.1.0</version>
  </dependency>
</dependencies>
```

[The repository URL should be entered as a single line. —*Ed.*]

I am going to create the following three classes. All these classes are included in the code bundle for this article.

- `MessageActionListener` implements the `org.eclipse.paho.client.mqttv3.IMqttActionListener` interface. I use instances of this class to specify the callback that will run code whenever a message has been successfully published to a topic.
- `Drone` represents a drone that has a name and can send and receive messages through the MQTT broker. This class not only encapsulates the data and logic related to drones as well as the messages and the commands included in messages, but it also implements the `MqttCallback` and `ImqttActionListener` interfaces defined in `org.eclipse.paho.client.mqttv3`. There is no type in these interface names—the naming convention for interfaces in the Java library is a bit confusing because some interfaces start with `I` while others don't. I use `Drone` instances as callbacks for specific events.
- `MqttSample01` creates three instances of the `Drone` class, makes them connect to the MQTT broker, and sends messages and commands. This class declares the `main` static method for the example application.

Creating a Class That Is Notified When Asynchronous Actions Are Complete

The asynchronous API requires you to work with callbacks. In this example, I demonstrate many ways of working with the necessary callbacks. The following code shows the import statements and the code for the `MessageActionListener` class that implements the `IMqttActionListener` interface.

```
import
  org.eclipse.paho.client.mqttv3.IMqttActionListener;
import
  org.eclipse.paho.client.mqttv3.IMqttToken;
public class MessageActionListener
implements IMqttActionListener {
    protected final String messageText;
    protected final String topic;
    protected final String userContext;

    public MessageActionListener(
      String topic,
      String messageText,
      String userContext) {
        this.topic = topic;
        this.messageText = messageText;
        this.userContext = userContext;
    }

    @Override
    public void onSuccess(
      IMqttToken asyncActionToken) {
        if ((asyncActionToken != null) &&
             asyncActionToken.getUserContext()
               .equals(userContext))
        {
         System.out.println( String.format(
           "Message '%s' published to topic '%s'",
           messageText, topic));
        }
      }

    @Override
    public void onFailure(
        IMqttToken asyncActionToken,
        Throwable exception) {
            exception.printStackTrace();
        }
}
```

When I create an instance of the `MessageActionListener` class, I need to specify the topic to which the message is going to be published, the message text, and a user context. The constructor saves the received values in immutable fields that have the same names as the received arguments.

What does it mean that a message was successfully delivered by the MQTT broker? It depends on the quality of service (QoS) that you select.

The class implements the `onSuccess` method required by the `IMqttActionListener` interface. Whenever an instance of the `MessageActionListener` class is used as a callback for an asynchronous action, the Java client invokes the `onSuccess` method when the action has been completed successfully, and it passes an asynchronous action token (`asyncActionToken`) of type `IMqttToken` as an argument to specify the action that has been completed. The method makes sure that `asyncActionToken` is not null and checks whether the value returned by `asyncActionToken.getUserContext()` matches the `userContext` saved by the constructor. If they match, the successful event is related to the event I wanted to monitor for its successful execution, and the code displays a message containing the message text that has been published and the name of the destination topic. I use an instance of this class as a callback for each message that is published, and thereby I can see all the successfully published messages.

The class also implements the `onFailure` method, which is required by the `IMqttActionListener` interface and simply calls the `printStackTrace` method for the received exception.

Specifying the Quality of Service

What does it mean that a message was successfully delivered by the MQTT broker? It depends on the quality of service (QoS) that you select when you work with the MQTT protocol. The QoS level is the agreement between the publisher and the receiver of a message about the guarantees for delivering the message. Delivering a message involves publishing from the client to the broker and then from the broker to the subscribed client. MQTT supports three possible QoS values:

- Level 0 means *at most once*: This level provides the same guarantee as the TCP protocol. The message is not acknowledged by the receiver. The sender neither stores nor redelivers any messages. As you might expect, this level has the lowest overhead.
- Level 1 means *at least once*: This level provides a guarantee that the message will be delivered at least once to the receiver. The main drawback is that this QoS level might generate duplicates, because the message can be delivered more than once. The sender stores the message until it receives an acknowledgment. In the event the acknowledgment isn't received within a specific time, the sender will publish again.
- Level 2 means *exactly once*: This level provides a guarantee that the message is delivered only once to the receiver. This QoS level makes sure the message isn't delivered more than once and, therefore, there is no chance for duplicates. However, as you might expect, it has the highest overhead because it requires two flows between the sender and receiver (one to receive, the other to send acknowledgment of receipt). Only when the entire flow is completed is the message considered to be successfully delivered.

In this example, I will work with QoS level 2, because I don't want the possibility of receiving a command twice. Messages are delivered even across network and client restarts. However, for that to occur, each message needs to be stored in a safe location until it has been successfully delivered. The Java client works with a pluggable persistence mechanism to store the messages. To keep things simple in this example, I will use memory-based persistence, which is not the best

option with QoS level 2. However, you can easily explore and configure other pluggable persistence mechanisms to avoid losing messages in the event the application, the JVM, the computer, or the device running the application stops working or shuts down.

Creating a Class to Represent a Drone That Processes Messages

The following code shows the key aspects of the `Drone` class that implements the `MqttCallback` and `IMqttActionListener` interfaces. The full class is available in the downloadable code.

When I create an instance of the `Drone` class, it is necessary to specify the desired `name` for the drone. The class defines many constants that I will use throughout the code and determine the string that defines a command key, the separator, the command that retrieves the altitude for the drone, the topic to which the Java client will subscribe, the desired QoS level, and the encoding that will be used for the messages, as shown below.

```
public class Drone implements MqttCallback,
    IMqttActionListener {
    public static final String COMMAND_KEY = "COMMAND";
    public static final String COMMAND_SEPARATOR = ":";
    public static final String
        GET_ALTITUDE_COMMAND_KEY = "GET_ALTITUDE";
    // Replace with your own topic name
    public static final String TOPIC =
        "java-magazine-mqtt/drones/altitude";

    public static final String ENCODING = "UTF-8";

    // Quality of Service = Exactly once
    // I want to receive all messages exactly once
    public static final int QUALITY_OF_SERVICE = 2;
    protected String name;
    protected String clientId;
    protected MqttAsyncClient client;
    protected MemoryPersistence memoryPersistence;
    protected IMqttToken connectToken;
    protected IMqttToken subscribeToken;

    public Drone(String name) { this.name = name; }

    public String getName() { return name; }
```

It is very important that you replace the `TOPIC` string with your own, unique topic name. The Mosquitto broker I am using in the example is public and, therefore, I need to use a unique topic to make sure I receive only the messages published by my code. I have specified `"java-magazine-mqtt/drones/altitude"` for `TOPIC` in this example. MQTT uses topic names that have a hierarchy and are separated by a slash (/). Another example of a topic name is `"java-client/samples/drones/commands/altitude"`.

The `connect` method has some code that is commented out just to remind you that in a production environment you shouldn't send messages over an insecure connection and your MQTT broker should work with TLS/SSL and require the appropriate authentication. The code creates an instance of the `MemoryPersistence` class to use as the previously explained pluggable persistence, generates a unique client ID, and creates an instance of `MqttAsyncClient` named `client`. This way, I create the entry point for the Java client with the asynchronous API.

```
public void connect() {
    try {
        MqttConnectOptions options =
            new MqttConnectOptions();
        // options.setUserName(
        //    "replace with your username");
```

```
// options.setPassword(
//    "replace with your password"
//    .toCharArray());
// Replace with ssl:// and work with TLS/SSL
// best practices in a
// production environment
memoryPersistence =
    new MemoryPersistence();
String serverURI =
    "tcp://iot.eclipse.org:1883";
clientId = MqttAsyncClient.generateClientId();
client = new MqttAsyncClient(
                serverURI, clientId,
                memoryPersistence);
// I want to use this instance as the callback
client.setCallback(this);
connectToken = client.connect(
                options, null, this);
    } catch (MqttException e) {
        e.printStackTrace();
    }
}

public boolean isConnected() {
    return (client != null) &&
           (client.isConnected());
}

@Override
public void connectionLost(Throwable cause) {
    // The MQTT client lost the connection
    cause.printStackTrace();
}
```

The line that calls the `setCallback` method uses `this` as an argument because I use the actual instance as the callback that will execute specific methods when some asynchronous events occur. The `setCallback` method requires an argument of the `MqttCallback` type. The `Drone` class implements the `MqttCallback` interface that requires the following three methods: `connectionLost`, `messageArrived`, and `deliveryComplete`. I'll get back to these methods later. It is very important to call the `setCallback` method before establishing the connection with the MQTT broker.

The line that calls the `client.connect` method specifies `this` as the last argument because I will also use the actual instance as the callback that will execute specific methods when some asynchronous events related to the connection occur. The fourth argument for the `connect` method requires an argument of the `IMqttActionListener` type.

The `Drone` class implements the `IMqttActionListener` interface that requires these two methods: `onSuccess` and `onFailure`.

```
@Override
public void onSuccess(
              IMqttToken asyncActionToken) {
        if (asyncActionToken.equals(connectToken)) {
            System.out.println( String.format(
                "%s successfully connected",name));
            try {
                subscribeToken = client.subscribe(
                    TOPIC, QUALITY_OF_SERVICE,
                    null, this);
            } catch (MqttException e) {
                e.printStackTrace();
            }
        }
        else if (asyncActionToken.equals(
                        subscribeToken))
        {
            System.out.println( String.format(
```

```
                "%s subscribed to the %s topic",
                name, TOPIC));
            publishTextMessage( String.format(
                "%s is listening.", name));
        }
    }

@Override
public void onFailure(IMqttToken asyncActionToken,
                      Throwable exception)
{
    // The method will run if an operation failed
    exception.printStackTrace();
}
```

I implemented the same interface in the `MessageActionListener` class. However, in this case, I will implement the interface to run code when the success or failure is related to the connection, not with messages as happened in the `MessageActionListener` class.

The code saves the `IMqttToken` returned by the `client.connect` method to the `connectToken` protected field. This way, I am able to check whether the `onSuccess` method's execution is related to this token or not. The connection uses asynchronous execution, and the `onSuccess` method for the `Drone` class will be executed after the connection with the MQTT broker has been successfully established.

The `onSuccess` method displays a message indicating that the specific drone has been successfully connected. Then, the code calls the `client.subscribe` method with the topic to which I want to subscribe and the desired QoS level. The call to this method specifies `this` as the last argument because I will also use the actual instance as the callback that will execute specific methods when some asynchronous events related to the subscription occur. The fourth argument for the `subscribe` method requires an argument of the `IMqttActionListener` type. So, after a successful subscription, the Java client will run the same `onSuccess` method, but the code will recognize that the event is related to the subscription because the received token won't match the connection token and instead will match the subscription token. It is not necessary to make it this way. It is possible to create an anonymous type to declare the methods that are necessary for the subscription callback for the asynchronous subscription. However, I wanted to demonstrate the usage of the tokens.

The `onSuccess` method displays a message indicating that the specific drone has been successfully subscribed to the specific topic. Then, the code calls the `publishTextMessage` method to publish a message to the topic indicating that the drone is listening.

```
public MessageActionListener publishTextMessage(
            String messageText)
{
    byte[] bytesMessage;
    try {
        bytesMessage =
            messageText.getBytes(ENCODING);
        MqttMessage message;
        message = new MqttMessage(bytesMessage);
        String userContext = "ListeningMessage";
        MessageActionListener actionListener =
            new MessageActionListener(
                TOPIC, messageText, userContext);
        client.publish(TOPIC, message,
                userContext,actionListener);
        return actionListener;
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
    return null;
} catch (MqttException e) {
    e.printStackTrace();
```

```
            return null;
        }
    }
```

The `publishTextMessage` method receives a `String` in the `messageText` argument with the text message that the drone has to publish to the topic and returns an instance of the `MessageActionListener` class. The method calls the `messageTextgetBytes` method to generate a byte array that takes into account the UTF-8 encoding. The `MqttMessage` class requires a byte array with the message to be sent as an argument to create an instance. Then, the code creates an instance of the `MessageActionListener` class named `actionListener` and calls the `client.publish` method to publish the message on the topic with an `actionListener` instance as the last argument that specifies the desired callback. This way, the code declared in the `onSuccess` method for this class will run after the message has been successfully published to the specified topic.

The `publishCommand` method takes two `String` arguments: the command name and the destination name. The method uses the received values to build a command and then calls the previously explained `publishTextMessage` method with this command as an argument. For example, the following command requests a drone whose name is `Drone #1` to print its altitude in feet: `COMMAND:GET_ALTITUDE:DRONE #1`.

```
public MessageActionListener publishCommand(
          String commandName, String destinationName)
{
     String command = String.format("%s%s%s%s%s",
         COMMAND_KEY, COMMAND_SEPARATOR,
         commandName, COMMAND_SEPARATOR,
         destinationName);
     return publishTextMessage(command);
}
```

Whenever a message arrives, the `messageArrived` method will be executed. The code in this method receives a `String` with the topic and an instance of `MqttMessage`.

```
@Override
public void messageArrived(String topic,
               MqttMessage message) throws Exception
{
    // A message has arrived from the MQTT broker
    // The MQTT broker doesn't send back
    // an acknowledgment to the server until
    // this method returns cleanly
    if (!topic.equals(TOPIC)) {
        return;
    }

    String messageText =
        new String(message.getPayload(), ENCODING);
System.out.println( String.format(
    "%s received %s: %s", name, topic,
    messageText));
String[] keyValue =
    messageText.split(COMMAND_SEPARATOR);
if (keyValue.length != 3) {
    return;
}
if (keyValue[0].equals(COMMAND_KEY) &&
    keyValue[1].equals(
        GET_ALTITUDE_COMMAND_KEY) &&
        keyValue[2].equals(name))
    {
        // Process the "get altitude" command
        int altitudeInFeet = ThreadLocalRandom
            .current().nextInt(1, 6001);
        System.out.println( String.format(
            "%s altitude: %d feet",
```

```
                    name, altitudeInFeet));
            }
    }
```

First, the code makes sure that the topic is the one that the drone is interested in. Then, the code calls the `message.getPayload()` method to retrieve the byte array with the message. The code creates a `String` instance with the byte array and the UTF-8 encoding as arguments to generate the appropriate `String`. The code displays a message indicating that the specific drone has received a message in the topic. Finally, the code processes the `String` with the received message to determine whether the message is a command. If it has the command-key prefix, the code decodes the command and if it's the "get altitude" command, the code displays a message with a pseudorandom altitude value for the drone expressed in feet.

Whenever a message has been successfully delivered, the `deliveryComplete` method is executed with a token (`token`) of type `IMqttDeliveryToken` as an argument to allow you to identify which message has been delivered. As previously explained, the meaning of a successfully delivered message will depend on the QoS level. In this case, I'm working with QoS level 2 and this method will be executed after the acknowledgment from the receiver arrives. Here, I didn't add code to this method and I just declared it to implement all the methods required by the interface.

```
    @Override
    public void deliveryComplete(
                    IMqttDeliveryToken token) {
        // Delivery for a message has been completed
        // and all acknowledgments have been received
    }
```

Publishing Messages

The following code lines show the code for the `MqttSample01` class that declares the `main` method. The `main` method creates three instances of the Drone class—`drone1`, `drone2`, and `masterDrone`—and calls the `connect` method for each of these instances. I didn't use any kind of list to work with the drones because I wanted the code to be easier to read. Forgive me for repeating some code in this `main` method.

```
public class MqttSample01 {
    public static void main(String[] args) {
    Drone drone1 = new Drone("[Drone #1]");
    drone1.connect();
    Drone drone2 = new Drone("[Drone #2]");
    drone2.connect();
    Drone masterDrone = new Drone("*Master Drone*");
    masterDrone.connect();

try {
    while (true) {
        try {
            Thread.sleep(5000);
            int r =
                ThreadLocalRandom.current()
                    .nextInt(1, 11);
            if ((r < 5) && drone1.isConnected()) {
                masterDrone.publishCommand(
                    Drone.GET_ALTITUDE_COMMAND_KEY,
                    drone1.getName());
            } else
            if (drone2.isConnected()) {
                masterDrone.publishCommand(
                    Drone.GET_ALTITUDE_COMMAND_KEY,
                    drone2.getName());
            }
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } catch(Exception e) {
            e.printStackTrace();
        } finally {
            if (drone1.isConnected()) {
                try {
                    drone1.client.disconnect();
                } catch (MqttException e) {
                    e.printStackTrace();
                }
            }
                ...similarly for drone2 and masterDrone…
```

```
*Master Drone* successfully connected
[Drone #2] successfully connected
[Drone #1] successfully connected
[Drone #2] subscribed to the java-magazine-mqtt/drones/altitude topic
[Drone #1] subscribed to the java-magazine-mqtt/drones/altitude topic
*Master Drone* subscribed to the java-magazine-mqtt/drones/altitude topic
Message '[Drone #1] is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
Message '*Master Drone* is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
Message '[Drone #2] is listening.' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
[Drone #2] received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
[Drone #2] received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: [Drone #2] is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: *Master Drone* is listening.
*Master Drone* received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: [Drone #1] is listening.
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
Message 'COMMAND:GET_ALTITUDE:[Drone #1]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #1] altitude: 3746 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
Message 'COMMAND:GET_ALTITUDE:[Drone #2]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
[Drone #2] altitude: 4224 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #2]
Message 'COMMAND:GET_ALTITUDE:[Drone #1]' published to topic 'java-magazine-mqtt/drones/altitude'
[Drone #1] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #2] received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
[Drone #1] altitude: 433 feet
*Master Drone* received java-magazine-mqtt/drones/altitude: COMMAND:GET_ALTITUDE:[Drone #1]
```

Figure 1. Console messages

```
        }
    }
```

Then, a forever-running loop generates a pseudorandom number every five seconds and, based on that number, it makes `masterDrone` publish a command to get the altitude for either `drone1` or `drone2`.

Figure 1 shows an example of the output messages shown in the Console window of the IDE. Notice that both Drone #1 and Drone #2 receive the same messages with the `GET_ALTITUDE` command. However, only the drone that is the destination for the message processes the command and displays its pseudorandom altitude.

Conclusion

This example demonstrates how you can use the Eclipse Paho Java Client and a Mosquitto MQTT broker to subscribe to a topic and publish messages to a topic. There is also a Java client library that can run on Android, in case you need to work with MQTT in Android. Whenever you need to exchange messages with an asynchronous, nonblocking API, you can consider using MQTT and the Java client. `</article>`

Gastón Hillar (@gastonhillar) has been working as a software architect with Java since its first release. He has 20 years of experience designing and developing software. He is the author of many books related to software development, hardware, electronics, and the Internet of Things, and he has been awarded the Intel Black Belt Software Developer Award eight times.

learn more

- The MQTT protocol
- The iot.eclipse.org project

Exploring Compact Profiles

Java 8's Compact Profiles point the way to Java 9's modularity.

BEN EVANS

Java 8 introduced the concept of Compact Profiles, which are reduced versions of the Java runtime environment (JRE) that do not contain the usual full contents of `rt.jar`. In this article, I explore the advantages of using Compact Profiles and how they point the way toward a modular future for the JDK, which takes a big step forward in Java 9.

The current versions of the JRE are quite monolithic: `rt.jar` is at least 60 MB on its own, without taking into account the size of native code loaded as dynamic libraries. If we can reduce the size of the Java platform footprint and move to a modular view of the JDK, we can realize some great benefits:

- Faster JVM startup times
- Reduced resource consumption
- Removal of packages that, in hindsight, shouldn't be in the core
- Improved security, because removing unused classes reduces the attack surface of the platform
- Convergence of the Java ME Connected Device Configuration (CDC) with Java SE

The initial approach to this effort was a full modularization of the platform, known as Project Jigsaw. This ambitious project also included other goals:

- Incorporate best practices for dependencies into the platform core, and apply lessons learned about dependency management from tools such as Maven, Apache Ivy, OSGi standard, and Linux distributions.
- Isolate dependencies—solve the library versioning problem.
- Allow application developers to package their code as modules, rather than as JAR files.

The catch is that in order to achieve the full set of goals, major surgery on the Java platform core is required. In particular, a new approach to classloading—involving a "modular classloader"—is required. This has a lot of edge cases and is a complex undertaking, especially if we need to maintain backward compatibility.

In order to maintain the release date for Java 8, the decision was made to move full modularity out to Java 9, which will ship in July. Java 8 Compact Profiles are designed to be a first step toward full modularity, with some of the basic advantages noted above. They build on the initial work done for modularity and provide a cut-down Java runtime, which

- Is fully compliant with the JVM and Java language specifications
- Has a substantially reduced footprint
- Removes functionality that is not always needed (for example, CORBA)
- Works for many applications (especially server-side code)

Compact Profiles are based on packages; they contain a number of full packages, and no partial packages are currently allowed. They are also subject to two other restrictions:

- A profile must form a closed set; references to classes not contained in the profile are not allowed.
- If a profile contains some classes from another, smaller profile, it must contain all of them, so partially overlapping profiles are not allowed. Put another way, profiles are additive.

PHOTOGRAPH BY JOHN BLYTHE

The smallest Compact Profile is called `compact1`, and it comprises the following packages: `java.io`, `java.lang`, `java.lang.annotation`, `java.lang.invoke`, `java.lang.ref`, `java.lang.reflect`, `java.math`, `java.net`, `java.nio`, `java.nio.channels`, `java.nio.channels.spi`, `java.nio.charset`, `java.nio.charset.spi`, `java.nio.file`, `java.nio.file.attribute`, `java.nio.file.spi`, `java.security`, `java.security.cert`, `java.security.interfaces`, `java.security.spec`, `java.text`, `java.text.spi`, `java.time`, `java.time.chrono`, `java.time.format`, `java.time.temporal`, `java.time.zone`, `java.util`, `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`, `java.util.function`, `java.util.jar`, `java.util.logging`, `java.util.regex`, `java.util.spi`, `java.util.stream`, `java.util.zip`, `javax.crypto`, `javax.crypto.interfaces`, `javax.crypto.spec`, `javax.net`, `javax.net.ssl`, `javax.script`, `javax.security.auth`, `javax.security.auth.callback`, `javax.security.auth.login`, `javax.security.auth.spi`, `javax.security.auth.x500`, and `javax.security.cert`.

Two other Compact Profiles are specified in Java 8: `compact2`, which adds packages used for remote method invocation (RMI), SQL, and XML, and `compact3`, which comprises all of `compact2` plus tooling and management packages (including Java Management Extensions [JMX]) as well as additional cryptography libraries. The smallest profile, `compact1`, occupies around 11 MB, which is a significant space savings.

As currently specified, all of these profiles are headless; they do not contain any GUI classes. Any applications requiring GUI support (Swing or AWT) must use a full JRE.

Tools

To make use of Compact Profiles, developers require tools. One important question is whether an application can run against a specific profile. Java 8 ships with two tools that have been enhanced to help answer this question: both `javac` and `jdeps` have been modified to be aware of profiles.

`javac` is the tool of choice for determining whether a collection of source code can be safely run on a specific profile. This is achieved by using the new `-profile` switch. `javac -profile <profile>` will cause a compilation error to be generated for any usage of a class not present in the indicated profile.

In some cases, however, source code is not available or a recompilation run is inconvenient. Fortunately, in this case, the new `jdeps` tool can help.

`jdeps` is a new static analysis tool that ships with Java 8. It provides an analysis of the dependencies of a specific class or JAR file. This tool is extremely useful (and not just for profile dependencies), and it features a `-profile` (or `-P`) switch that indicates which packages depend on which profiles.

Let's take a look at an example, which summarizes the package dependencies for the popular JUnit testing library. See **Listing 1**, which shows good news: everyone should be able to test their code.

Listing 1.

```
jdeps -s -P junit.jar
junit.jar                          -> compact1
```

If we want more information, we can use the `-v` switch for verbose output, which will give us a lot of detail about each class inside the JAR file. See **Listing 2** (some of the output was truncated because it was 2,152 lines long).

Listing 2.

```
jdeps -v junit.jar
junit.jar -> /Library/Java/JavaVirtualMachines/openjdk8/
Contents/
Home/jre/lib/rt.jar
   junit.extensions.ActiveTestSuite (junit.jar)
```

```
   -> java.lang.Class
   -> java.lang.InterruptedException
   -> java.lang.Object
   -> java.lang.String
   -> java.lang.Thread
   -> junit.extensions.ActiveTestSuite$1     junit.jar
   -> junit.framework.Test                   junit.jar
   -> junit.framework.TestCase               junit.jar
```

If we want a slightly more high-level view, we can use `-V package` to show dependencies between packages, as shown in **Listing 3** (some of the output was truncated because it was 1,297 lines long).

Listing 3.

```
jdeps -V package junit.jar
junit.jar -> /Library/Java/JavaVirtualMachines/openjdk8/
Contents/Home/jre/lib/rt.jar
   junit.extensions (junit.jar)
      -> java.lang
   junit.framework (junit.jar)
      -> java.io
      -> java.lang
      -> java.lang.annotation
      -> java.lang.reflect
      -> java.util
   junit.runner (junit.jar)
      -> java.io
      -> java.lang
      -> java.lang.reflect
      -> java.text
      -> java.util
```

`jdeps` is also very flexible about what it will accept as input: a JAR file, a directory, or a single `.class` file. It provides capabilities for recursive traversal and for specifying that only packages with a name that matches a given regular expression should be considered. It can also warn that code uses an internal API and is not portable between Java environments (and might break if run against a future version of Java).

Finally, let's look at the NetBeans IDE. The current version already has support for a wide range of JDK 8 features, including Compact Profiles. When selecting which JDK or JRE to use in Project Properties, for JDK 8 and later, a developer can choose whether to compile against the full JRE or a profile. This makes it much easier to ensure that when you're targeting a particular profile, unwanted dependencies don't creep in. With luck, other IDEs will follow suit and also add support to allow developers to write code in the IDE that checks conformance with a specific profile at development time.

A Word About Stripped Implementations

In addition to Compact Profiles, there was another technique that was proposed but ultimately not included in Java 8: Stripped Implementations. A Stripped Implementation was to be a reduced JRE, which was packaged with an application that had the exclusive use of it. Because the application was the only possible client for the Stripped Implementation, the runtime could be aggressively pruned, removing packages, classes, and even methods that were not used by the application.

This approach was advantageous in circumstances where resource limitations were severe. It relied on extensive testing to ensure that nothing that the application could rely on was removed by the stripping process. In general, it is extremely difficult to get a precise accounting of an application's dependencies. This is due in large part to the existence of techniques such as reflection and classloading, which can greatly complicate (or even render impossible) the task of ascertaining the true dependencies of a set of classes.

Compact Profiles are very different from Stripped Implementations—the former are Java runtimes designed

for general-purpose use and are complete implementations of the Java language specification, whereas Stripped Implementations were defined to be single-use, nonreusable implementations that did not have to conform to the Java language specification at their point of delivery to the user.

Stripped Implementations were targeted as a feature for Java SE 8, but due to some complications with licensing and the current testing kit, they had to be dropped very close to the release date.

Conclusion

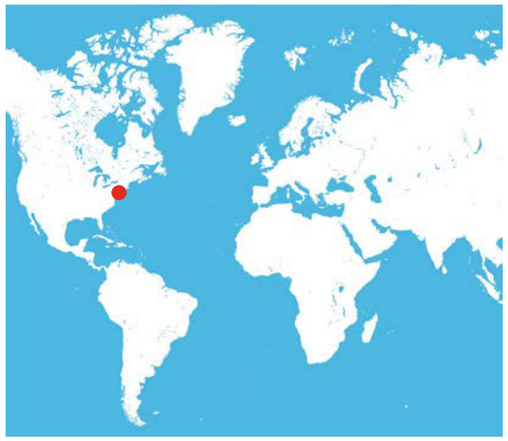
[Java 9, which ships midyear, implements a whole new modularity system based on Project Jigsaw. In most cases, that modularity system does away with the need for Compact Profiles and Stripped Implementations. This article, which originally appeared in 2014 and has been lightly updated, shows how the drive to smaller executables has been a focus of the Java team for a long time. Going forward, Compact Profiles will remain a useful option only for projects that need the reduced executables but, for one reason or another, cannot migrate to Java 9. —*Ed.*] </article>

Ben Evans (@kittylyst) is a Java Champion, tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee.

learn more

- Compact Profiles Demonstrated
- Compact Profiles overview
- Hinkmond Wong's EclipseCon 2014 presentation on Compact Profiles

THE PHILLY JUG



The Philadelphia Area Java Users Group (the Philly JUG) in Philadelphia, Pennsylvania, is one the world's oldest, largest, and most active JUGs. Operating continuously since 2000, the JUG has been recognized twice by Sun Microsystems as one of the top JUGs in the world. From its humble beginnings with 35 members sitting on the floor of an abandoned office, the JUG has grown to more than a thousand members. JUG meetings typically see 70 to 100 developers attending on a regular basis. Over the years, the JUG has hosted many Java luminaries, such as Neal Ford, Brian Goetz, Cameron Purdy, Rod Johnson, Gavin King, Marc Fleury, Greg Luck, Yakov Fain, and Kito Mann. The group has a strong roster of regular local speakers. You can follow the JUG's activities via its meetup group and its Twitter account.

Topics at JUG meetings are always of interest to the professional Java developer. Recent topics have included Java 9, Adopt-a-JSR, RxJava, Java on Azure, and NAO robots. The JUG has also been known to offer more-general technology concepts that have a Java (or JVM language) angle, such as Docker, Spark, and Lagom. The JUG engages speakers and topics to help keep the local software engineering community engaged and informed.

The Philly JUG aims to further expand its reach and continue to serve its local community. Among these efforts, the JUG is an active adopter of critical Java SE and Java EE JSRs, including Java EE 8, Servlet 4, and Java SE 9. The JUG is an active partner member of the Java Community Process.

PETER LEDBROOK

Gradle's Java Library Management

As builds become more complex, library dependencies present a special challenge that build tools are working to solve.

Before the days of Apache Maven and Apache Ivy, making use of third-party libraries was a laborious affair. Every time you wanted to include a library, you would have to download manually not only its JAR but also any other JARs that it depended on. You would then go through a similar process whenever you wanted to upgrade a library. Tracking down the necessary transitive dependencies could be particularly painful, as could identifying any dependencies you no longer needed.

Both Maven and Ivy simplified the process by allowing you to declare which dependencies you needed via a set of standard coordinates: a group, a name, and a version. The tools took on the responsibility of locating and downloading the necessary JARs for you—a process known as *dependency resolution*. Make no mistake, automatic dependency management (in addition to a single public Java library repository—Maven Central) fundamentally changed the way Java developers worked.

This approach has been refined over the years, but what you are using now is still at heart the same. In fact, you might be forgiven for thinking that dependency management is a "solved" problem.

Automatic dependency management introduced its own set of issues that can bedevil developers. Build tools can and should do more to help, because they are best placed to improve the situation. If you're interested in learning what that help looks like, then read on as I investigate several common challenges of working with Java libraries—both third-party and your own.

Consuming Third-Party Libraries

In an ideal world, you would declare your project's dependencies and everything would just work. But this rarely happens except in small projects. A more likely outcome is that you would encounter one or more of the following issues:

- An unexpected library version that breaks the build or your app/library
- Multiple versions of a library on a classpath
- Multiple JARs with the same class or classes
- Unwanted or unnecessary dependencies
- Dependency resolution that works for some developers but not others

The sources of these issues are varied, ranging from poorly defined metadata to changes in the names of dependencies. And don't underestimate the inherent challenge of keeping a system working that relies on perhaps hundreds of moving parts, with each library having its own release schedule, time constraints, priorities, and so on.

Given that you're likely to encounter problems with dependency resolution (unless you never change or upgrade any of your dependencies), it's crucial that you be able to identify the underlying issues quickly. That's why I think developers deserve better diagnostic tools.

Diagnosing Dependency Issues

You might think that tracking down the source of a version conflict or some other dependency issue is simply a case of looking at the dependency graph. You can do this with most

build tools, but remember that dependency graphs can be large and complex. Trying to identify the source of a problem in such scenarios is often elusive and frustrating—even if you know what you're looking for.

Imagine you have added a library to your Hibernate 4–based project that, without you realizing it, has pulled in Hibernate 5. If you're lucky, your project continues to work regardless of which version of Hibernate the build tool selects. Alternatively, the build or the running project might fail in a way that makes it obvious there's an issue with the version of the Hibernate library. In that case, you can run Gradle's `dependencyInsight` command (which is a built-in task) to find out which dependencies transitively depend on Hibernate.

What if your project falls into neither category? What if, instead, the build or running project fails in a distinctly unhelpful way? In that case, the obvious question to ask is: what changed? Unfortunately, build tools haven't made it easy to answer that question in the past.

One workaround is to disable automatic conflict resolution, resulting in the build tool reporting an error if there is more than one version of a library in the dependency graph. You can do this in Gradle by adding the following snippet to your build file:

```
configurations.all {
    resolutionStrategy {
        failOnVersionConflict()
    }
}
```

This approach unfortunately forces you to resolve even minor version conflicts yourself. It also fails to help if a library's name or group has changed.

Wouldn't it be great if you could just ask the build tool to show the differences between the dependency graphs of the current build and the last working one? This feature is available in Gradle Enterprise, saving developers effort and time in diagnosing many dependency issues.

Identifying the source of a dependency issue is only part of the solution, because you still need to fix it somehow.

Ideally, what you need is some way to declare whether a dependency is exposed in your component's API or not.

Fixing Dependency Issues

Build masters and developers traditionally resolve dependency problems using exclusions, a solution that reminds me of the adage that when all you have is a hammer, everything looks like a nail. Exclusions may solve the problem in practice, but they're often fragile and time-consuming to maintain.

What you need is an expressive way to specify rules or constraints for the dependency resolution engine. Taking the earlier Hibernate example, what you probably want to say is "always use version 4.3.11.Final of Hibernate." You can do this with Gradle using the following build script snippet:

```
configurations.all {
  resolutionStrategy {
    force
      'org.hibernate:hibernate-core:4.3.11.Final'
  }
}
```

This ensures that only the specified version of the `hibernate-core` dependency is included on your project's classpath.

You might also want to consider what happens if Hibernate 3 or 5 appears in the dependency graph. Do you simply want to force their versions as previously described? Or does it make more sense to add a warning or even fail the build

because the libraries that depend on those versions are unlikely to work with a different major version? You can implement any of these approaches via Gradle's resolution strategy API.

Managing library dependencies at the build level is deceptively hard.

There is no single correct strategy for dependency resolution that works for everyone, which is why Gradle's API is so flexible. But most of the time, you probably won't want to deal with it at a fine-grained level. Developers at Netflix have open-sourced two plugins that help by providing higher-level abstractions. The Nebula Resolution Rules Plugin enables powerful, declarative, centralized resolution rules across your whole organization, while the Nebula Dependency Lock Plugin solves the issues of build instability and staging when using snapshot versions and version ranges (known as *dynamic versions*).

Third-party libraries are only part of the story. Most sites also build and reuse their own libraries, which I discuss next.

Writing and Consuming Your Own Libraries

With the advent of multiproject or multimodule builds, it has never been easier to break your systems into several components. Your build tool manages the dependencies between the individual components automatically, avoiding any manual installation or publishing steps.

Intermodule dependencies generally work well, but there are still some important issues to consider. One of them is how to deal with libraries outside of the multimodule build—a topic tackled in the previous section. Another is a fundamental flaw in the way dependencies are specified. For example, consider the module relationships in **Figure 1**.

The arrows represent compile-time requirements, and each box is a module you own, with Joda Time being the only third-party library. In this case, `date-utils` needs `joda-time` in order to compile, which you would specify in Gradle as

```
dependencies {
    compile 'joda-time:joda-time:2.9.7'
}
```

This seems simple enough, so what's the problem? The flaw is that this simple dependency graph ignores the consumer's perspective. It doesn't answer the question: does `core` (the consumer of `date-utils`) need `joda-time` in order to compile, or is the date-utils module enough?

Build tools currently play it safe and always include `joda-time` as a compile-time dependency of `App`. But this simply pollutes the compilation classpath of `core` if `date-utils` doesn't expose any Joda Time types in its public API—that is, if the use of `joda-time` is an internal implementation detail.

Ideally, what you need is some way to declare whether a dependency is exposed in your component's API or not. You will be able to do so from Gradle 3.4 onward. We are introducing two new configurations (or *scopes*, in Maven terminology) for Java projects: `api` and `implementation`. As part of the new Java library plugin, these configurations will allow you to express the effect of dependencies on consumers.

Going back to the example in **Figure 1**, imagine that `date-utils` does expose Joda Time types in its public API. Then you would declare the dependency in your `date-utils` build script this way:

```
dependencies {
    api 'joda-time:joda-time:2.9.7'
}
```

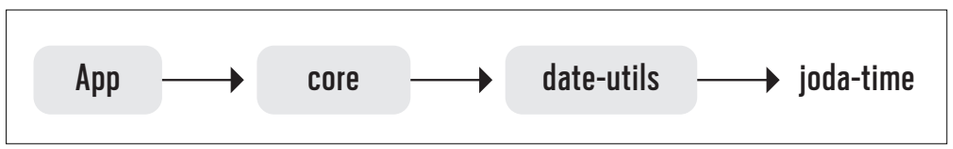


Figure 1. Module dependencies in a multimodule build

By declaring `joda-time` as an API dependency, you ensure that it is included on the compilation classpath of `core`, because `core` requires `date-utils` to compile.

Now consider the relationship between `date-utils` and `core` from the perspective of `App`. Does `App` need to be aware of `date-utils` at compile time? In this case, let's assume that `date-utils` is used purely internally by `core`. You would express this in `core`'s build script this way:

```
dependencies {
    implementation project(':date-utils')
}
```

What these two dependency declarations mean for the overall project is

- `core` has both `date-utils` and `joda-time` on its compilation classpath
- `App` has `date-utils` and `joda-time` on its runtime classpath but not its compilation classpath
- `App` requires `core` at compile time

This approach has several significant advantages:

- JARs don't leak onto compilation classpaths when it's unnecessary for them to do so.
- Changing an `implementation` dependency doesn't force a recompilation of consumers.
- Published POMs match the requirements of consumers.
- The relationships between components are clearer.

With respect to the published POM, Gradle uses a simple mapping of `api` to `compile` and of `implementation` to `runtime`. This mapping makes sense because a transitive `api` dependency is required in order to compile the consumer, but that's not the case for a transitive `implementation` dependency.

I've discussed consuming libraries so far, but only in the context of dependency repositories and multimodule builds. There is one other scenario to consider: when you want to work from the source code version of a library that is not part of your multimodule build.

Improving Versioned-Library Workflows

There is an interesting debate going on at the moment regarding the relative merits of a *monolithic repository* (monorepo), which is a single source code repository for *all* your code, versus the more usual design, which relies on separate repositories for different projects. Of particular relevance to this discussion of libraries is one of the declared benefits of a monorepo: easier cross-project changes.

Consider this workflow:

- You discover that the project you're working on requires a fix to a library it uses.
- You ask that library's team to implement the fix.
- You incorporate the fixed library into your project.

In the case of a monorepo, you have access to the fixed version as soon as you update your local working copy or repository. And if you deploy off the trunk/master, you can deploy a new version of your project with the fix right away.

The process is less straightforward if the library is in a different source repository. The typical solution involves getting the library maintainer's team to publish a new version of the JAR, often as a snapshot. But snapshots are notoriously unreliable because the actual binary can change at any moment. So there is no guarantee that a given fix will appear in the next release version.

Another option is to check out or clone the source repository for the library and build it locally. Then you become less dependent on the library's team to publish updated versions. On the other hand, you still end up relying on locally installed or published snapshots—unless you are able to make your project depend on the library's source project, and then build them together. Adding a dependency on the source distribution (rather than just an evolving

binary) would give you the following advantages:

- You control exactly what code you're working against, via a commit ID, tag, or some other label.
- You can make fixes yourself and test them without any intermediate steps.
- You don't need to rely on snapshots that can easily break your project again.
- You can easily test in-progress work by updating your working copy/repository.

You can achieve this with Gradle's *composite builds* feature. It enables you to incorporate any other Gradle build into your own, dynamically replacing the corresponding declared dependencies.

To understand how it works, think back to **Figure 1** and imagine you need a fix for the Joda Time library. Also imagine that it has a Gradle build. The maintainer is too busy to implement the fix, so you get hold of the source code project and put it alongside or even inside your project's root directory. Then you can apply the necessary changes yourself.

Now, rather than building `joda-time` separately and installing or publishing its JAR, you can run

```
./gradlew --include-build ../joda-time-copy build
```

This will automatically build the project at `../joda-time-copy` and put its JAR on the compilation classpath of `date-utils` when it builds that, even though the latter declares a versioned dependency of `joda-time`. Even better, your tests will run against this development version of `joda-time` as well. This approach greatly improves the development cycle for cross-project changes and works particularly well with the IDE support in IntelliJ and Eclipse, which uses the Buildship plugin.

One other consideration is how you share your work with other team members without having to publish intermediate versions of the library for them to use. It's an awkward problem, but you can solve it by persisting the build inclusion definition in your project's `settings.gradle` file. This is explained on the Gradle blog. Just be aware that you should use something like Git submodules or Subversion externals to ensure that your team members have a local copy of the library's source when they build your project. Alternatively, you can make the inclusion conditional on the library existing locally.

It's also worth thinking about how you manage the continuous integration of your own libraries and applications that reside in separate source repositories. With Gradle, you can create a composite build for your continuous integration servers that includes only the builds you need—it's otherwise empty—and runs all the integration tests. Both the application and the library teams then get immediate feedback on any breaking changes without having to publish intermediate binaries.

Composite builds smooth the development process and buy time for teams to coordinate releases where necessary. While this discussion involved separate repositories, you can also use this feature with a monorepo if you want to use that architecture while retaining independently versioned libraries.

Conclusion

Managing library dependencies at the build level is deceptively hard, as anyone who has worked extensively with them will attest. In this article, I've presented several of the challenges that Java developers and build masters face. Many of you have probably encountered at least some of these in your own day-to-day work. The ultimate goal of the build tool developer is to make dependency management truly manageable. </article>

Peter Ledbrook is an independent consultant who has maintained many builds over the years, including ones based on Make, Ant, and Maven. He now does training and technical writing for Gradle Inc.

FEATURED JAVA SPECIFICATION REQUEST

JSR 372: JSF 2.3

JavaServer Faces (JSF) 2.3, which is governed by JSR 372, recently ended its public comment period. This event lets us see what is new in mostly a minor update from version 2.2. It includes the addition of incremental features that the community requested. The community has driven JSF 2.3 very heavily, directly committing many of the features into the codebase of the reference implementation, Mojarra. These are the features slated to be in JSF 2.3:

- Alignment with the Java SE 8 Date and Time API
- Improved CDI support
- Formal deprecation of the JSF-specific bean subsystem in favor of CDI
- WebSocket integration
- Ajax method invocation
- Multifield validation

Besides these changes, there are many other smaller updates. For details on these, check out the specification document itself. There is a useful change list at the very beginning of the PDF document. The community has been doing a good job blogging about JSF 2.3 features—particularly folks such as Arjan Tijms and Anghel Leonard. In addition, the July/August 2016 issue of *Java Magazine* had a detailed explanation of the new features, written by Tijms, who is on the expert group for this JSR.

Download the draft specification from the JCP site. Do your part by engaging actively. Try out the new version and provide comments on the new features.

[Thanks to Reza Rahman for his contributions to this write-up. —*Ed.*]

Quiz Yourself

Intermediate and advanced test questions

SIMON ROBERTS

As promised in my last column, with this issue, I begin to include questions that simulate the level of difficulty of the Oracle Certified Associate exam, which contains questions for a more preliminary level of certification than the questions that regularly have appeared here.

Of course, I also include the more advanced questions that simulate those from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

Because there is little value in asking easy questions, I provide only intermediate and advanced questions (and they are marked as such). These levels correlate with the two exams. However, don't let the "intermediate" tag fool you—the questions are never trivial.

Question 1 (intermediate). Given this code (with line numbers at left), which is a fragment of a larger method body:

```
14: StringBuilder[] sba = {
15:   new StringBuilder("Fred"),
16:   new StringBuilder("Jim"),
17:   new StringBuilder("Sheila")
18: };
19:
20: System.out.println("sba[2] is " + sba[2]);
21:
```

Which two of the following are true?

a. The array referred to by `sba` might be eligible for garbage collection at line 19.
b. The array referred to by `sba` might be eligible for garbage collection at line 21.
c. Assigning `sba = null;` at line 21 would make the array referred to by `sba` and the three `StringBuilder` objects definitely eligible for garbage collection.
d. The array referred to by `sba` and the three `StringBuilder` objects will definitely be eligible for garbage collection when the method returns to its caller.
e. The array referred to by `sba` and the three `StringBuilder` objects might not be eligible for garbage collection even after the method returns to its caller.

Question 2 (intermediate). Given the following code:

```
// line n1
switch (x) {}
```

Which two of the following lines of code can be added successfully at line n1? Assume that `x` has no declaration in scope at line n1 and assume that each line is added individually.

a. `boolean x = false;`
b. `short x = 99;`
c. `int x = 0;`
d. `long x = 0;`
e. `StringBuilder x = new StringBuilder("x");`

Question 3 (advanced). Given this:

```
public class Recorder {
  class Record {}
  class LongRecord extends Record {}

  public List<LongRecord> gatherRecords() {
    return Arrays.asList(
      new LongRecord(), new LongRecord()
    );
  }

  public void processRecords(
      Collection<Record> records) { // line n1
    records.forEach(System.out::println);
  }

  public void gatherAndProcess() {
    List<LongRecord> lr = gatherRecords(); // line n2
    processRecords(lr);
  }

  public static void main(String[] args) {
    new Recorder().gatherAndProcess();
  }
}
```

Which is true? Choose one.

a. The code produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

b. Changing the argument type declaration at line n1 to `List<Record>` produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

c. Changing the argument type declaration at line n1 to `Collection<? super Record>` produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

d. Changing the argument type declaration at line n1 to `Collection<? extends Record>` produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

e. Changing the variable type declaration at line n2 to `List<Record>` produces output in the following form:

```
Recorder$LongRecord@1218025c
Recorder$LongRecord@816f27d
```

Question 4 (advanced). Given this code:

```
List<String> ls = Arrays.asList("Fred", "Jim", "Sheila",
    "Fred");
Set<String> s1 = new HashSet<>(ls); // line n1
Set<String> s2 = new TreeSet<>(ls); // line n2
System.out.println(s1.equals(s2));
```

What is the result? Choose one.

a. Line n1 causes compilation to fail.
b. Both line n1 and line n2 cause compilation to fail.
c. An exception is thrown at line n1.
d. Outputs `true`.
e. Outputs `false`.

Question 1. The correct answers are options C and E. The basis on which an object is eligible for garbage collection is simply that it cannot be reached from any live thread. In other words, the program can never get hold of the object again. At line 19, `sba` is an entirely valid reference to the original array. Consequently, `sba` is definitely not eligible for collection at that point. That's really important because if it were collected, the attempt to print `sba[2]` on line 20 would have unpredictable results. Generally, Java avoids any unpredictable behaviors (concurrent programming is the significant exception to this). This means that option A is incorrect.

Similarly, although `sba` does not appear to be used again at or after line 21, it is definitely not eligible for garbage collection because the reference is still valid, and it could be used later in the method. Note that the question explicitly states that there is more to the method. Therefore, option B is also incorrect.

However, if line 21 executes `sba = null;`, the previous value of the reference is overwritten. Because the now-lost value in `sba` was the *only* reference to the array created in lines 14 through 18, and that value was never stored anywhere else (not even as an argument to a method call) between its creation and line 21, you know that there are now zero references to that object that are available to the program. As a result, the array is definitely unreachable and is eligible for garbage collection. Also, references to the three `StringBuilder` objects were written only into the array, and because the array is not reachable in the program, neither are the `StringBuilder` objects, and they, too, are eligible for garbage collection. Because of this, option C is correct.

Options D and E are, perhaps obviously, the inverses of one another, so one must be true and the other false. As a side note, exam creators try to avoid this sort of situation —because it makes a question easier—but sometimes it is unavoidable, so it is worth spotting inverse answers. Be sure your logic is sound, though. For example, if option E had said "definitely not eligible," the two would not have been inverses of each other, because the important possibility of "don't know" would be a third possibility not covered by either.

In this case, because one or the other must be true, either you can definitely collect the objects at the return of the method or you cannot be sure. Which is it? Generally, local variables go out of scope and cease to exist when a method returns. This would suggest that the objects referred to by those variables become eligible for garbage collection. However, in this case, you don't see the whole method body. If a reference to the array "escapes" from the method, you can make no such assumption. At least three mechanisms exist by which this escape might occur, the most obvious of which is probably when the method itself returns the value of `sba` to its caller. A second possibility is that this method stores the value of `sba` in a variable that has greater visibility and longer life, such as a static variable or collection. In addition, passing `sba` as a method argument might allow the value to escape, because you cannot know if that method stores the value somewhere in a manner similar to the previous point. As a result, you cannot know if the array and, therefore, the three `StringBuilder` objects are eligible for garbage collection. Consequently, option E is true and option D is false.

Question 2. The correct answers are options B and C. The rules for a `switch` statement require that the argument be assignment-compatible with an int (allowing for autounboxing), an enum, or a String. *Java Language Specification* section 14.11 states that the type of the expression "must be

char, byte, short, int, Character, Byte, Short, Integer, String, or an enum type, or a compile-time error occurs." String and enum types haven't always been allowed; enum was new with Java 5. Therefore, it didn't apply before that and, in fact, String was not legal until Java 7.

Although String is a legal type for a `switch` statement, `StringBuilder` is not; neither is it assignment-compatible with String.

Because of these rules, options A, D, and E are all incorrect, while both options B and C are correct.

Question 3. The correct answer is option D. This question delves into one of the most startling consequences of the type-erasure mechanism of Java's generics system. When a variable of a collection type is declared with its generic specification—as in something like `List<Record>`—the "Record" part of the information exists only at compile time. The underlying object is still a `List` that actually accepts any object type. The power of generics is that they allow the compiler to do "consistency checking" that can ensure that type errors cannot happen in the code.

As part of the consistency checking, the compiler verifies that all assignments are safe from the perspective of the Liskov Substitution Principle. This is a principle in object orientation that requires that if you assign `b = a`, `a` must be capable of fully substituting for `b`. That is, all the behaviors expected of the type of `b` must be properly implemented, and work as expected, in the object referred to by `a`.

Now, the trick here is that, in fact, the compiler does not accept that assigning a `List<LongRecord>` to a `Collection<Record>` is a valid substitution. Because the assignment that occurs in the invocation of the method declared around line n1 is not valid, the code does not compile and option A is incorrect.

This question really hinges on why that assignment is not valid, and how you might correct that problem. Option B suggests that by changing the argument type from `Collection` to `List`, you might fix it. However, a `List` is fully capable of substituting for a `Collection` (the `List` interface extends `Collection`, and all the methods are implemented). This part of the assignment is not the problem, and the change wouldn't alter the situation. Therefore, option B is incorrect.

Consider the following, because it's a little less confusing. The compiler thinks that a `List<LongRecord>` is not a valid substitute for a `List<Record>`. There's something that you can do with a `List<Record>` that isn't properly supported by a `List<LongRecord>`. That illegal operation is the addition of a `Record`. Think about that for a moment. A `List<LongRecord>` can properly contain anything that's assignment-compatible with a `LongRecord`: that would be `LongRecord` objects and any subclasses thereof. However, it should *not* contain any instances of the parent class `Record`. And it should not contain any sibling classes of `LongRecord`, if they existed. However, a `List<Record>` can legitimately contain such objects, and the compiler doesn't know that you don't add any in the method declared around line n1. Imagine the consequences if the method, prior to looping over the contents of `Collection`, invoked `records.add(new Record())`. That would be bad, right?

The code does not actually do any such terrible operations, but the compiler doesn't analyze that; it just knows that the code *could*. Fortunately, there's a syntax that lets you do what you want to do—that is, iterate over the contents of the collection, safely extracting things from it while knowing they're assignment-compatible with `Record`. In effect, you define that `Collection` can be a collection of "anything that's assignment-compatible with `Record`." Therefore, if `X` is a type that's assignment-compatible with `Y`, `X` must be "further down" the inheritance hierarchy (regardless of whether it is an object or an interface) than `Y`. In a sense, `X` extends `Y`. And that's how the syntax comes about; the language lets you say,

in effect, "collection of something," provided that something extends `Record`. (Note that in this sense, `Record` "extends" `Record` because it's assignment-compatible.) That syntax, of course, is `Collection<? extends Record>`. On that basis, option D is correct.

Option C is also interesting. In this situation, the proposed change would say that whatever `Collection` is intended to contain, `Record` must be assignment-compatible with that. If you presented a collection of `Object`, then `Object` is a superclass of `Record`, and you'd be able to *insert* the `Record` safely into `Collection`. However, you have no idea what types you might pull out of `Collection`, which might be a problem. But regardless of this, you still cannot pass `List<LongRecord>` into the method, because `List<LongRecord>` could not properly accept assignment of a `Record` into `List` even if you wanted to do that. Therefore, this syntax doesn't solve the problem at hand, and option C is incorrect. However, this is an important syntax when you want to assign things to the generic type of whatever is passed in.

Option E fails for the same reasons the code fails in its unchanged form. The return type of the `gatherRecords` method is `List<LongRecord>`, and that's not assignment-compatible with `List<Record>`. Therefore, changing the variable type on line n2 would fail to compile, even though it would allow the invocation of the method around line n1 to compile. Substituting one problem for another doesn't result in any output, though, so option E is incorrect.

Question 4. The correct answer is option D. Options A and B suggest compilation failures while attempting to construct and initialize the sets. This might happen if there is no constructor available that accepts a `List` as an argument. However, both `Set` types provide such a constructor. The documentation for collections calls for collection implementations to provide constructors that accept an argument of type `Collection`, and as a result, a `List` is valid. Because of this, both options A and B are false.

Option C suggests that an exception is thrown during construction of the `Set`. This might be plausible, given that the `List` has a duplicate entry, which is not permitted in a `Set`. However, the documentation notes that "all constructors must create a set that contains no duplicate elements." The effect is that both `Set` objects contain only the three distinct elements, `"Fred"`, `"Jim"`, and `"Sheila"`. It's also fair to observe that, in general, a `Set` recognizes and ignores attempts to add duplicates, so an exception in this initialization situation would be unhelpful and counterintuitive. Because no exception is thrown, option C is false.

As a side note, it's possible for a `TreeSet` to throw an exception during the addition of items, if the items being added to it are not `Comparable` and no suitable `Comparator` has been provided. In this case, however, the items being added are String objects, and String implements `Comparable<String>` as needed, so no such problem arises.

Having eliminated all the other options, options D and E are essentially alternatives. The interesting thing here is that the documentation for the `equals` method of `Set` requires that all implementations return a value that indicates only whether the contents of the set are the same, without regard to the implementing class. As the documentation notes further, "This definition ensures that the `equals` method works properly across different implementations of the set interface." As a result, the output is the value `true`, and option D is correct, while option E is incorrect. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

//contact us /

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A-3133, Redwood Shores, CA 94065, USA.

- Subscription application
- Download area for code and other items
- *Java Magazine* in Japanese